1988

A conceptual level design for a static scheduler for hard real-time systems

O'Hern, Joanne T.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A CONCEPTUAL LEVEL DESIGN FOR A STATIC SCHEDULER FOR HARD REAL-TIME SYSTEMS

by

Joanne T. O'Hern

March 1988

Thesis Advisor Luqi

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified		1b Restrictive Markings	
2a Security Classification Authority		3 Distribution Availability of Report: Approved for public release; distribution is unlimited.	
2b Declassification Downgrading Schedule			
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)	
6a Name of Performing Organization: Naval Postgraduate School	6b Office Symbol (if applicable): 62	7a Name of Monitoring Organization: Naval Postgraduate School	
6c Address (city, state, and ZIP code): Monterey, CA 93943-5000		7b Address (city, state, and ZIP code): Monterey, CA 93943-5000	
8a Name of Funding Sponsoring Organization	8b Office Symbol (if applicable)	9 Procurement Instrument Identification Number	
8c Address (city, state, and ZIP code)		10 Source of Funding Numbers	
		Program Element No	Project No / Task No / Work Unit Accession No

11 Title (include security classification): A CONCEPTUAL LEVEL DESIGN FOR A STATIC SCHEDULER FOR HARD REAL-TIME SYSTEMS

12 Personal Author(s): Joanne T. O Hern

13a Type of Report	13b Time Covered	14 Date of Report (year, month, day)	15 Page Count
Master's Thesis	From To	March 1988	72

16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)
Field	Group	Subgroup	static scheduler, PSDL, computer aided rapid prototyping

19 Abstract (continue on reverse if necessary and identify by block number)

This thesis builds upon work previously done in the development of the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) and presents a conceptual design for the pioneer prototype of the static scheduler which is part of the CAPS execution support system. The design of hard real-time systems is gaining a great deal of attention in the software engineering field as more and more real-world processes are becoming automated. This increase in automation identified a need for the advancement of software design technology to meet the design requirements for these hard real-time systems. The CAPS and PSDL are tools being developed to aid the software designer in the rapid prototyping of hard real-time systems. PSDL, as an executable design language, is supported by an execution support system consisting of a static scheduler, dynamic scheduler, and translator. The static scheduler design includes the scheduling algorithms required to schedule time critical operators contained in a PSDL prototype in such a way that all operator timing constraints and precedence relationships are met to produce a feasible static schedule if one is possible. Implementation of the conceptual design will be the basis for further work in this area.

20 Distribution Availability of Abstract	21 Abstract Security Classification	
☒ unclassified unlimited ☐ same as report ☐ DTIC users	Unclassified	
22a Name of Responsible Individual: Luqi	22b Telephone (include Area code): (408) 646-2735	22c Office Symbol: 52Lq

DD FORM 1473,84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

Approved for public release; distribution is unlimited.

A Conceptual Level Design for a Static Scheduler
for Hard Real-Time Systems

by

Joanne T. O'Hern
Lieutenant, United States Navy
B.S., State University of New York at Albany, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN TELECOMMUNICATIONS SYSTEMS
MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
March 1988

ABSTRACT

This thesis builds upon work previously done in the development of the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) and presents a conceptual design for the pioneer prototype of the static scheduler which is part of the CAPS execution support system. The design of hard real-time systems is gaining a great deal of attention in the software engineering field as more and more real-world processes are becoming automated. This increase in automation identified a need for the advancement of software design technology to meet the design requirements for these hard real-time systems. The CAPS and PSDL are tools being developed to aid the software designer in the rapid prototyping of hard real-time systems. PSDL, as an executable design language, is supported by an execution support system consisting of a static scheduler, dynamic scheduler, and translator. The static scheduler design includes the scheduling algorithms required to schedule time critical operators contained in a PSDL prototype in such a way that all operator timing constraints and precedence relationships are met to produce a feasible static schedule if one is possible. Implementation of the conceptual design will be the basis for further work in this area.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Rapid advances in computer hardware technology have made computer systems more available to perform everyday tasks. Over the past twenty years, hardware costs as a percentage of total system costs have decreased from 85% to about 15% [Ref. 1: p. 9]. Because computer hardware is becoming faster as well as cheaper, the demand for increasingly sophisticated computer applications is on the rise. In the Department of Defense (DoD), computers are being used to guide weapons systems, control satellites, and run communications networks. These applications are examples of embedded computer systems, they are only one part of larger overall systems. Although embedded systems have different applications, they do have several characteristics in common. They tend to be very large, require parallel processing, are subject to real-time constraints, and above all, must be highly reliable [Ref. 1: p. 3]. Such systems are also referred to as hard real-time systems. Hard real-time systems are characterized by timing constraints that absolutely must be met.

Computer technology is an integral part of today's telecommunications systems. Embedded computers control message processing, routing, and switching subject to hard real-time or near hard real-time constraints in such systems as the Naval Communications Processing and Routing System (NAVCOMPARS), Naval Modular Automated Communications Subsystem (NAVMACS), the Common User Digital Information Exchange Subsystem (CUDIXS), The Automated Digital Network (AUTODIN), the Defense Data Network (DDN), the Defense Switched Network (DSN), and MILSTAR.

The development of software for these large systems is very time consuming and costly. On the average, one software programmer produces ten lines of code per day. Embedded software systems typically range from thousands to millions of lines of code [Ref. 1: p. 15]. This labor intensive software development now accounts for approximately 90% of the cost of a computer system [Ref. 1: p. 9]. Currently, there are no existing computer aided systems available to assist the designer in the critical earlier stages of development of large software systems with hard real-time constraints.

The remainder of this chapter is an introduction to software engineering, describing software design methodologies and a set of proposed software tools for aiding designers with the early stages of developing hard real-time systems.

A. SOFTWARE ENGINEERING AND DESIGN METHODOLOGIES

Software engineering is the application of scientific and mathematical principles to the problem of making computers useful to people by means of software. It indicates the development of software that is modifiable, efficient, reliable, and understandable.

The traditional software life cycle and rapid prototyping are two of the more common design methodologies used to maintain a scientific approach to software engineering.

1. Traditional Software Life Cycle

This methodology, also known as the traditional waterfall, is shown in Figure 1 on page 3. The traditional lifecycle consists of seven phases in the development of software products. These phases are outlined briefly below [Ref. 2]:

1. Requirements Analysis - this phase establishes the purpose of the proposed software system.
2. Functional Specifications - a model of the proposed system is constructed. This model only contains those aspects of the system that are visible to the users.
3. Architectural Design - a model of the implementation is constructed. The software modules and their interfaces that will be used to realize the system are identified.
4. Module Design - during this phase of development, the algorithms and data structures that will be used to realize the behavior specified in the architectural design will be chosen.
5. Implementation - executable programs are produced, usually in a high level programming language.
6. Testing - in this phase, faults will be detected by running programs with selected input data.
7. Evolution and Repair - new features/capabilities are added onto the system and necessary design changes are made to repair faults.

A major problem associated with using this methodology in the design of large real-time systems is that there is no guarantee that the resulting product will be reliable or even meet user specifications. The requirements of large systems are generally very difficult to describe. Often a user will only be able to indicate the true requirements by observing the execution of the system. The traditional software life cycle yields an executable program, especially in the case of large systems, only after too much time and money are spent.

2. Rapid Prototyping

An alternative methodology called rapid prototyping is proving to be much more efficient in the design of large real-time systems. The rapid prototyping

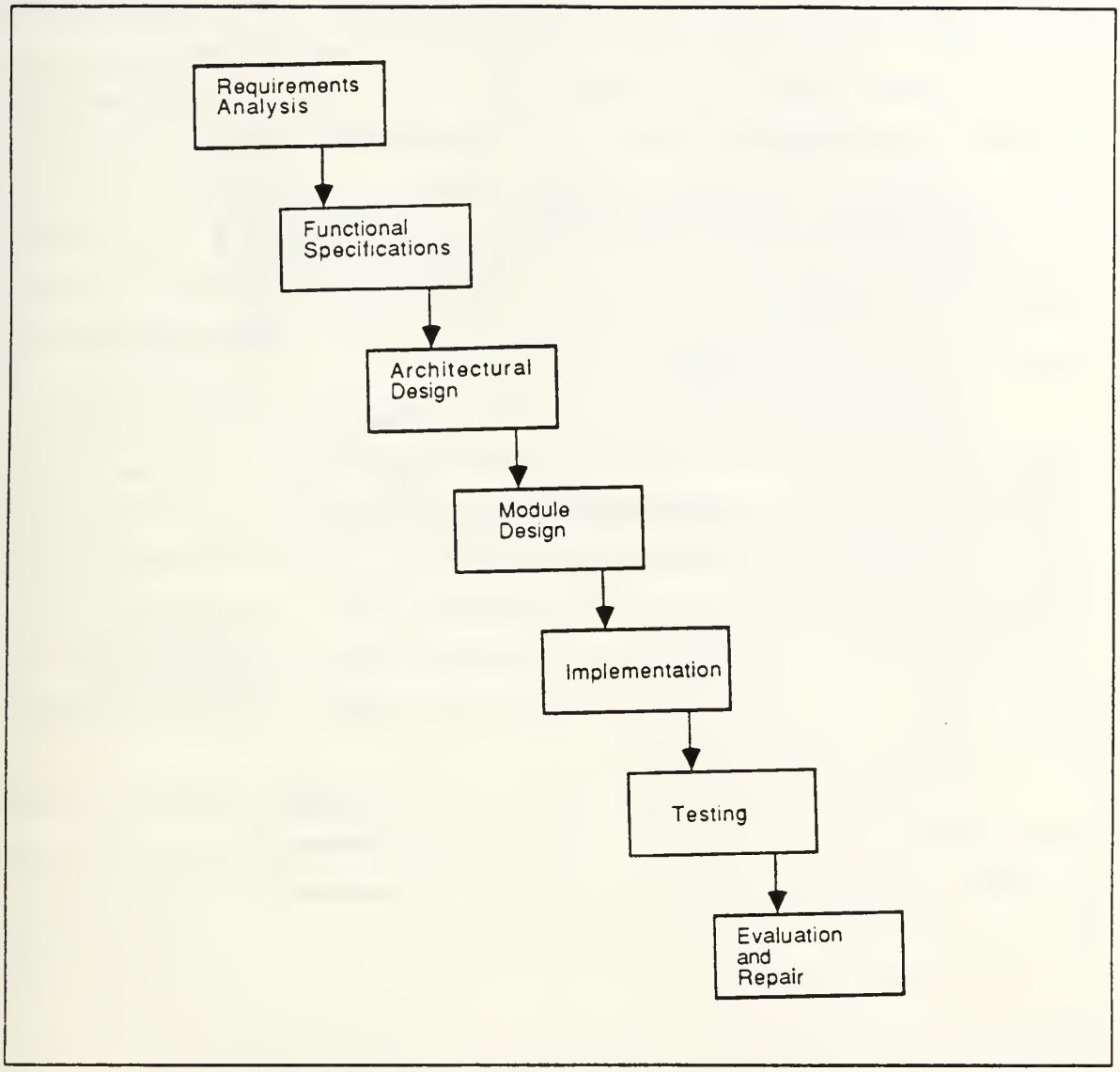


Figure 1. The Traditional Software Life Cycle Methodology

methodology is made up of two phases, 1) rapid prototyping and 2) automatic program generation. A prototype is an executable model of the intended system and is the product of the rapid prototyping phase. In general, the prototype is only a partial representation of the intended system and includes only the system's most critical aspects. The prototype must satisfy its requirements, be easy to modify, and be easy to read and analyze. In the rapid prototyping methodology, system requirements are determined and a prototype is constructed. The prototype will then be demonstrated to the user for requirement clarification and feasibility determination. Adjustments are

made as needed and the modified prototype is demonstrated. This iterative process is shown in Figure 2 on page 5 and continues until both the user and designer are satisfied that the system performs to user specifications. The rapid prototyping methodology generates an executable model faster and at less cost than the traditional software life cycle. In addition, the problems associated with a user being unable to accurately communicate his requirements to the designer are alleviated [Ref. 3: p. 3]. The final software product developed by using the rapid prototyping method should be more reliable, more in line with user needs, less expensive, and more timely than software developed using the traditional life cycle approach.

To date, rapid prototyping has been done manually without the aid of software tools. Each step in the rapid prototyping methodology, though faster than the traditional life cycle approach as discussed above, still requires a good deal of time and effort. In an attempt to make rapid prototyping more in line with its name, software tools are being developed to automate the rapid prototyping process. An automated rapid prototyping environment will enable the system designer to possibly retrieve software components from a software library and review previous designs at the touch of a button in addition to execution of the prototype.

At the present time, software methods are inadequate to handle the rapidly growing demand for large systems. "A significant improvement in software technology is needed to improve programming productivity and the reliability of software products" [Ref. 4: p. 66]. The Computer Aided Prototyping System (CAPS) is being developed to improve software technology, and will aid the software designer in the requirements analysis of large real-time systems by using specifications and reusable software components to automate the rapid prototyping process [Ref. 4: p. 66].

The Prototype System Description Language (PSDL) is an executable high level specification language that directly supports CAPS. PSDL is made executable by the execution support system element of CAPS. CAPS and PSDL will be described in greater detail in Chapter II.

B. OBJECTIVES

The objective of this thesis is the development of a conceptual level design for the design of the static scheduler for the prototyping language PSDL of the CAPS execution support system. The static scheduler design will be the basis for further research and implementation of the static scheduler.

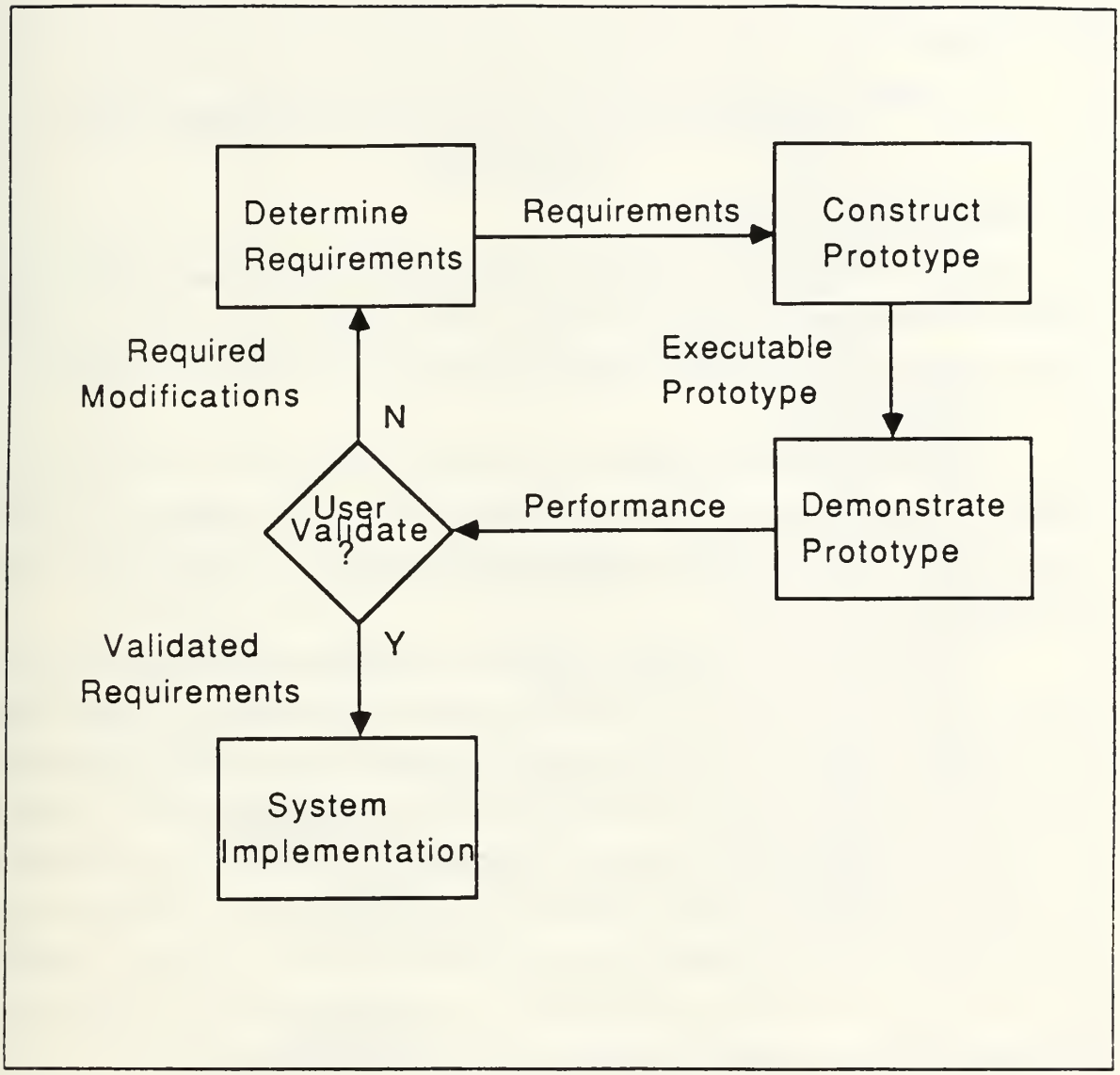


Figure 2. The Rapid Prototyping Methodology

C. ORGANIZATION

A survey of the state-of-the-art research into static scheduler designs and considerations will be presented in Chapter II. PSDL constructs and the conceptual level design for the static scheduler will be discussed in Chapter III. Conclusions and applications to telecommunications software development will be presented in Chapter IV.

II. PREVIOUS RESEARCH AND SURVEY OF SCHEDULING ALGORITHMS

A. PREVIOUS RESEARCH

The static scheduler for the Prototype System Description Language (PSDL) is one element of the Computer Aided Prototyping System (CAPS) tool. CAPS is presented here in greater detail to provide an overall framework for the PSDL static scheduler.

1. CAPS

Chapter I introduced CAPS as a tool that is being designed to aid software designers in the rapid prototyping of large software systems. CAPS makes use of specifications and reusable software components to automate the rapid prototyping methodology [Ref. 4: p. 66].

A base of reusable software components is searched based on module specifications entered by the designer. If a match is found, the component is retrieved from the component base for use in the prototype. If no match is found and the specification cannot be decomposed further, the designer must hand code the component. If further decomposition is possible the decomposed specifications are entered into the system and the library is searched once again. This process continues until all components have been retrieved or hand coded and is shown graphically in Figure 3 on page 7. Provided that it is in fact more efficient to accomplish the component search than to hand code each module, this tool should significantly increase software productivity.

The CAPS architecture contains the following elements:

1. User Interface
2. Prototyping System Description Language
3. Rewrite Subsystem
4. Software Design Management System
5. Prototype Data Base and Software Base
6. Execution Support System

These components will be discussed briefly below and are shown graphically in Figure 4 on page 8.

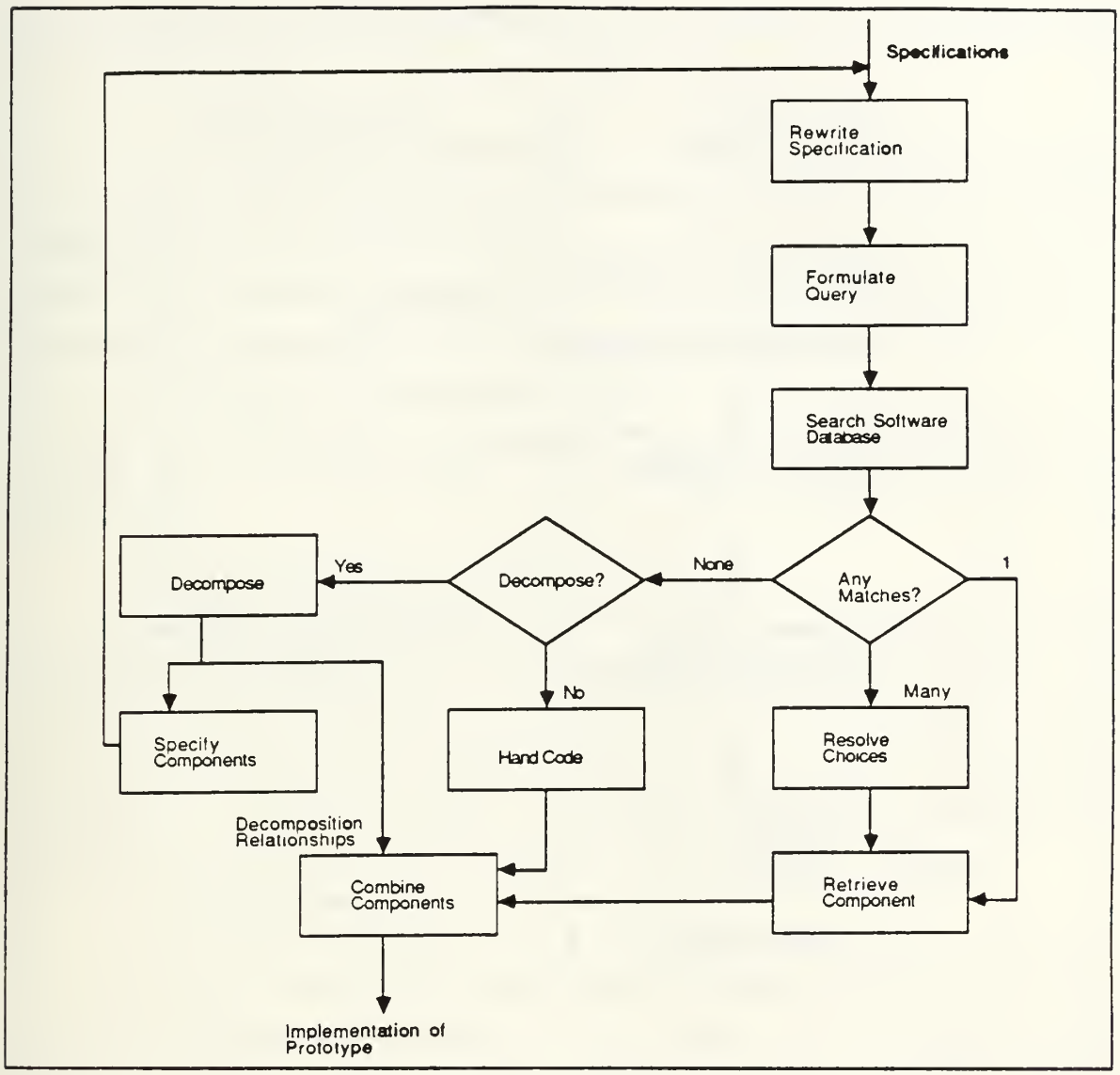


Figure 3. The Computer Aided Prototyping System Methodology

a. User Interface

The user interface consists of a syntax directed editor for PSDL and a graphics tool for constructing and displaying data flow diagrams [Ref. 5: p. 27]. The editor will automatically supply key words and provide legal syntactic alternatives for the designer to select at each step in the design.

b. Prototyping System Description Language (PSDL)

PSDL is designed specifically for use with the CAPS. It is intended to be used at the specification and design levels and supports functional, data, and control

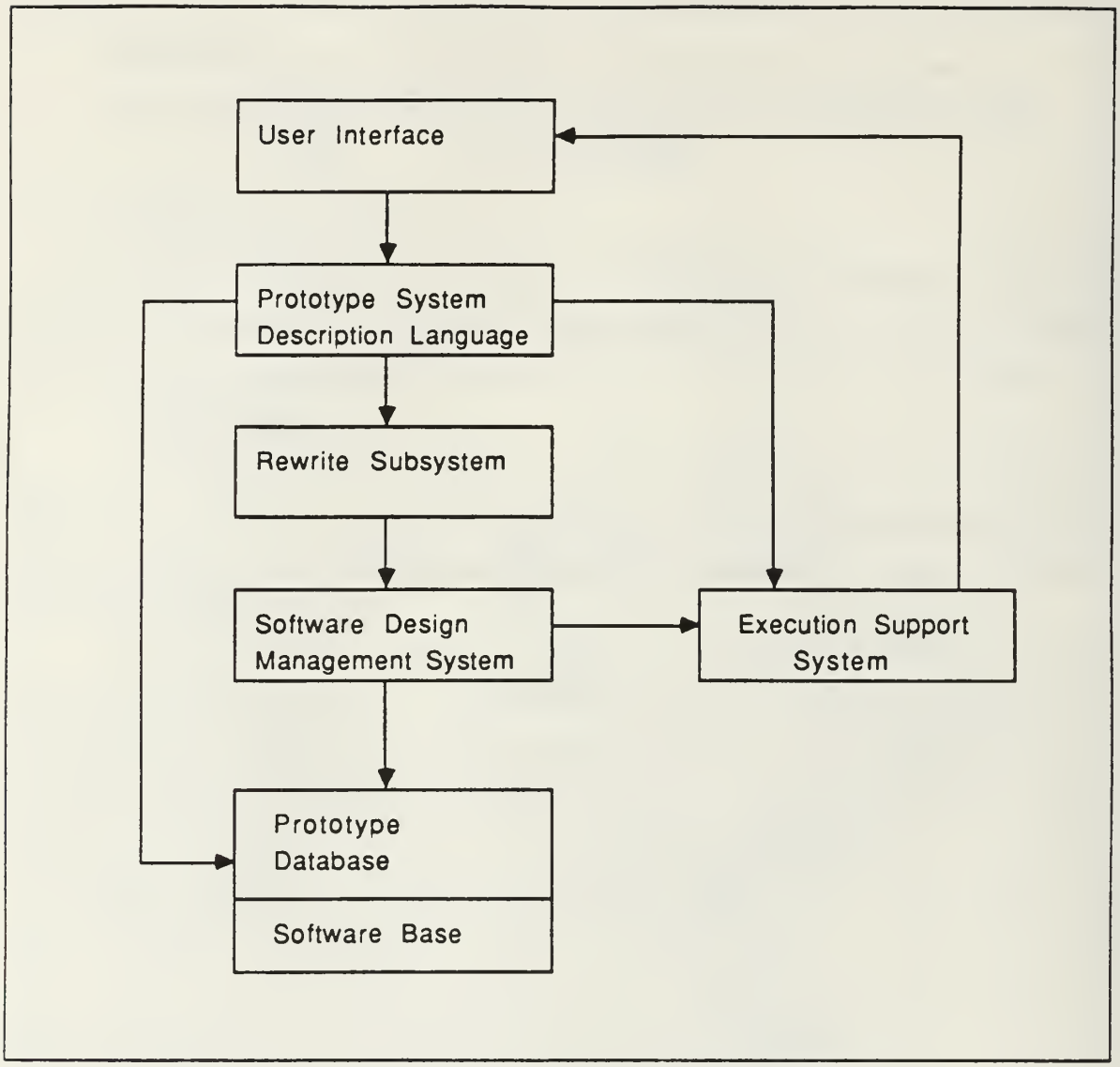


Figure 4. The Computer Aided Prototyping System Architecture

abstractions. The resulting PSDL prototypes are hierarchical decompositions based on both data flow and control flow.

PSDL is based on a computational model containing Operators that communicate via Data Streams. Operators may be either periodic or sporadic, functions or state machines, and atomic or composite. Data Streams are communications links connecting exactly two Operators and may be dataflow or sampled streams [Ref. 6: pp. 8-10]. PSDL does not contain any global or mutable data types, all data

must be passed to Operators via Data Streams. Those features of PSDL pertinent to the static scheduler will be discussed in more detail in Chapter III.

c. Rewrite Subsystem

The components contained in the software base are retrieved by searching the software base for matching specifications. Software designers can choose from several specifications but the software base is searched with one normalized specification. The function of the rewrite subsystem is to translate equivalent specifications (aliases) into the normalized specification (term). [Ref. 4: p. 69]

d. Software Base Management System

The software base management system is responsible for organizing, retrieving, and instantiating reusable software components from the software base. [Ref. 4: p. 69]

e. Prototype Data Base and Software Base

The prototype data base maintains copies of prototype structures and design information. The software base contains the reusable software components. The reusable components that make up the software base must be interpretable and portable. The component base itself will be easily extensible to allow the addition of new reusable components. It will also be possible for the designer to browse the component base to enable him to select and retrieve appropriate components. [Ref. 4: p. 70]

f. Execution Support System

The execution support system is necessary for the construction and updating of a prototype as well as prototype execution. The execution support system is made up of three components, a translator, a static scheduler, and a dynamic scheduler [Ref. 7: p. 7]. The translator translates the statements in the PSDL prototype into statements in an underlying programming language. The underlying programming language for the CAPS is Ada®.[1] The development of the translator is presented in Moffitt [Ref. 8]. The static scheduler attempts to find a static schedule for the operators in the PSDL prototype with real-time constraints. An implementation guide for the static scheduler can be found in Janson [Ref. 9]. The dynamic scheduler is a run time executive which controls the execution of the prototype, schedules operators that do not have real-time constraints, and provides facilities for debugging and gathering statistics. A design for the dynamic scheduler is contained in Eaton [Ref. 10].

---

1 Ada® is a registered trademark of the United States Government, Ada Joint Program Office.

The interfaces between these components are shown in Figure 5 on page 11. The designer interacts with the user interface to create PSDL source code. The dynamic scheduler initiates the actions of the translator and static scheduler and the PSDL source code is read directly by both. The translator translates the PSDL code into Ada source code and the static scheduler extracts operator timing information from the PSDL source code and creates a static schedule in Ada source code. The static scheduler provides the dynamic scheduler with the static schedule and the non-time critical operators. The Ada source code from the translator and the static scheduler is compiled, linked, and exported and the resulting executable Ada code is passed to the dynamic scheduler. In its debugging mode, the dynamic scheduler interfaces with the designer through the user interface.

B. SURVEY OF SCHEDULING ALGORITHMS

Research in the area of scheduling algorithms is becoming increasingly important as more and more real world processes are being performed by computers in embedded systems. The role of the static scheduler in the CAPS execution support system is very important to the execution of the prototype since a valid static schedule is a necessary (though not sufficient) condition for a successful prototype in terms of meeting its timing constraints. A task (or PSDL operator) is a software module that performs a particular function. In hard real-time systems, tasks or operators must be scheduled to meet critical timing constraints and failure to do so results in a failed system. The scheduling of tasks or operators can be divided into two separate scheduling problems:

1. Static scheduling which requires knowledge of all timing properties of tasks before scheduling begins, and
2. Dynamic scheduling which schedules tasks as they become known and does not know beforehand a task's timing properties.

In addition to the above distinction, static schedules tend to be inflexible and do not adapt well to an environment whose behavior is not completely predictable, though typically they have low run-time costs. In contrast, dynamic schedules are flexible and adapt easily to changes in the environment but tend to have higher run-time costs. The static scheduler being designed for PSDL will be able to handle unpredictable environments by creating equivalent periods for unpredictable (sporadic) operators. The operators are scheduled based on these equivalent periods so that whenever they do occur, there will be a time slot available within an acceptable time frame.

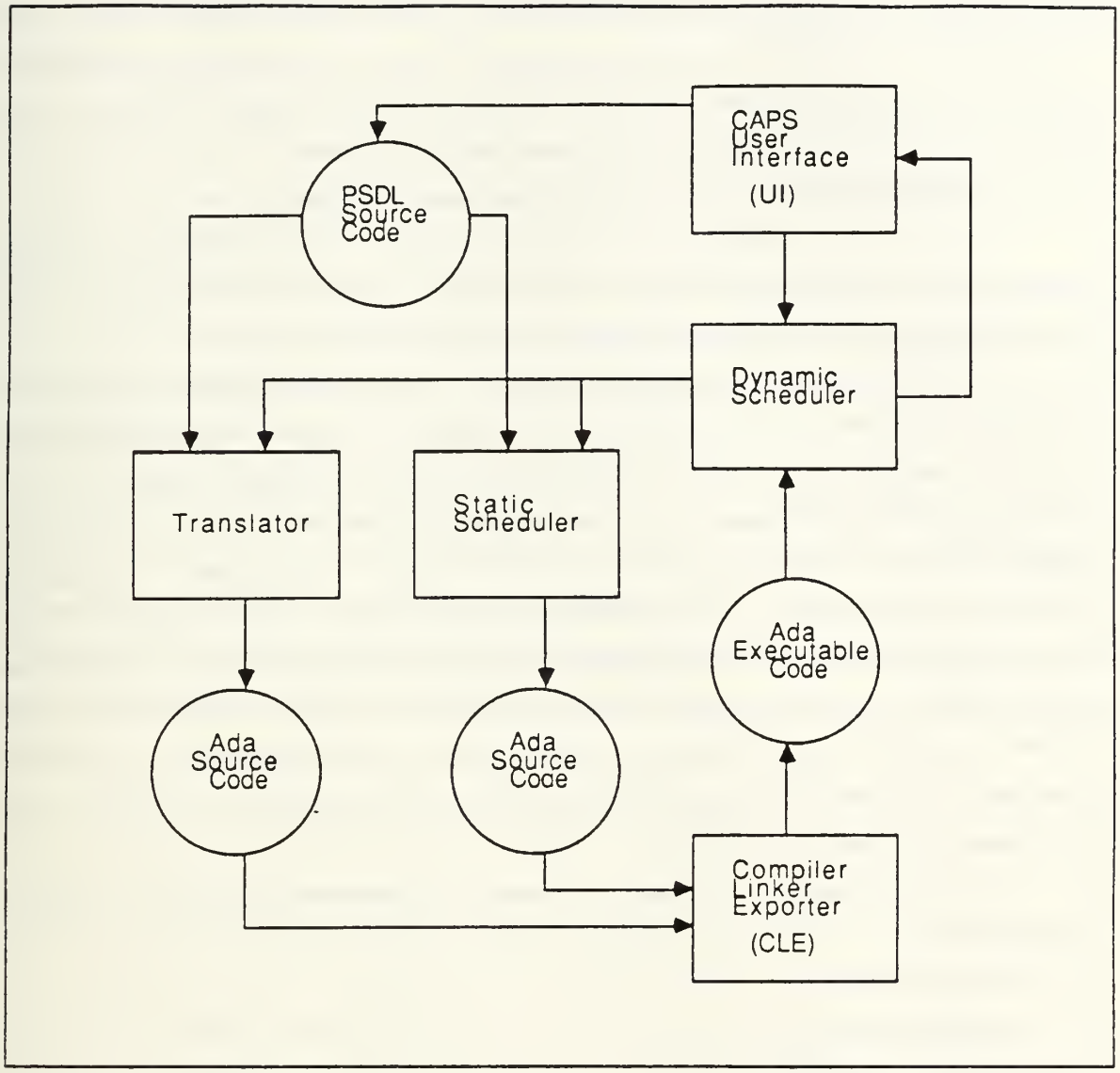


Figure 5. The Execution Support System

Both static and dynamic schedules can be built for single or multiprocessor systems. A multiprocessor system can be centralized (each processor shares a common memory) or distributed (each processor has an independent memory).

This section briefly describes some of the research efforts that are being directed toward static scheduling algorithms for both single and multiprocessor systems. In addition to the work being done on CAPS and PSDL, researchers are looking at additional ways to automate the software design environment for real-time systems. Static scheduling algorithms are but one important aspect of this automation effort.

This chapter is divided into three sections. In the first section, several system decomposition and static scheduling algorithms currently being designed are presented. The second section looks at aids in determining software safety. The third section presents work on monitoring the execution of real-time systems.

1. System Decomposition and Static Scheduling Algorithms

One of the most important aspects in real-time system development is expressing the critical timing constraints of a hard real-time system and decomposing systems into models that facilitate the scheduling of the time critical tasks.

Dasarathy, [Ref. 11], explores constructs for expressing real-time timing constraints for systems modeled as finite_state machines. Performance constraints and behavioral constraints are identified as two categories of timing constraints in hard real-time systems. Performance constraints set limits on the response time of the system while behavioral constraints set limits on a user's stimuli to the system. To completely model a real-time system, both of these types of constraints must be included. Three types of temporal constraints are maximum, minimum, and durational. Maximum timing constraints stipulate an upper bound between the occurrence of two events. Conversely, minimum timing constraints stipulate a lower bound between the occurence of two events. Durational timing constraints require that an event must occur for a specified amount of time.

Both maximum and minimum timing requirements include four types of constraints:

1. Stimulus - Stimulus
2. Stimulus - Response
3. Response - Stimulus
4. Response - Response

Cases 1 and 3 are constraints posed on the users of a system and can be expressed in a design language by using timers. Cases 2 and 4 are constraints on the system's performance. In a maximum time situation, these constraints can be expressed by using a latency statement and in a minimum time situation, a delay statement is appropriate. Latency specifies the maximum amount of time that can pass between the two events and delay specifies the minimum amount of time between two events.

Constructs for expressing durational timing constraints typically include a duration attribute specifying the length of the duration of the stimulus or response. If this

attribute is not specified, the event is considered to be instantaneous, requiring virtually no time.

The Prototype System Description Language contains constructs for expressing all of these types of constraints. Both human and hardware properties can be simulated when demonstrating the prototype to indicate the feasibility of the constraints imposed on the behavior of users and equipment.

In addition to expressing the system's timing constraints, a real world system needs to be decomposed into both its time critical and non-time critical tasks. The order of execution of the tasks then must be controlled, usually by one of two types of control commonly found in hard real-time systems; data flow and control flow. Data flow systems can be modeled as directed graphs where nodes represent operators and arcs show the data dependencies among operators. An operator is fired when all incoming data arrives on its input arcs. The operator consumes these values and then produces an output value which is an input for another operator. Computation continues to proceed in this data-driven manner. Control flow systems rely on a centralized control such as a program counter or main module that determines when an operator should fire.

Prior to developing static scheduling algorithms to schedule the time critical tasks or operators of a hard real-time system, the timing constraints and relationships among tasks must be specified such that these constraints and relationships can be adhered to. Mok and Sutanthavibul present a graph model for describing real-time systems and their timing constraints and relationships where execution is governed by the availability of data (dataflow) [Ref. 12]. The authors study those real-time systems where input data arrive at fixed rates (periodic systems) but otherwise there are no explicit timing constraints.

The graph model is a triple (G, f, h) where G = (V, E) is a directed acyclic graph. An example of an instance of the graph model is shown in Figure 6 on page 14. V is the set of nodes and E is the set of directed edges. Nodes represent either input elements or computation elements. Input elements have no incoming edges, they only represent the source of periodic data. The second element of the triple is a function mapping each node into a list of integer parameters. Computation nodes are mapped onto a non-negative integer which is its computation time and input nodes are mapped onto a pair of integers, the time when the node can first be implemented (lag) and the length of the time interval between two successive requests for the input node (period).

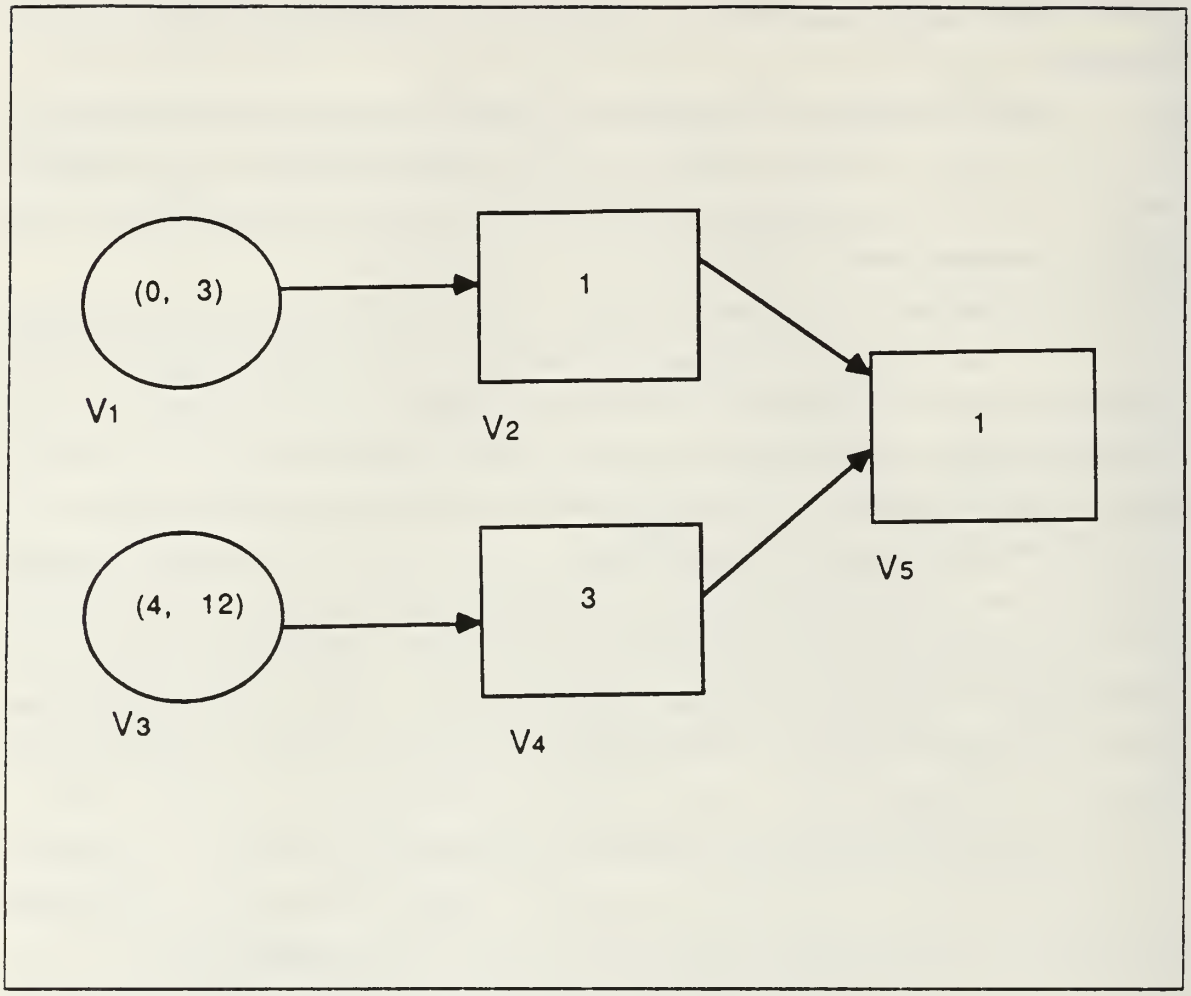


Figure 6. Example of an Instance of the Graph Model

The third element in the model, h, maps each edge onto a non-negative integer denoting the size of the data item passing between two nodes. In the single processor case, communication times are assumed to be negligible and can be ignored.

The scheduling problem is dependent on whether non-preemptive or preemptive scheduling is to be used. In non-preemptive scheduling, a task is uninterruptible and must run to completion once it is started. In preemptive scheduling, tasks may be interrupted prior to completion. It was found that preemptive scheduling tends to yield a greater number of valid schedules than does non-preemptive scheduling. Despite this finding, the PSDL static scheduler to be presented in Chapter III utilizes non-preemptive scheduling.

Mok and Sutanthavibul then describe their strategy for transforming the nodes in the graph model into a set of independent processes in a process model. This strategy results in an efficient on-line scheduler when preemption is allowed.

An instance of the process model consists of a set of processes each of which can be scheduled independently. This implies that there are no precedence constraints or other contraints on the execution of the processes. This differs from the PSDL static scheduler concept in that the processes (known as operators in PSDL) are constrained by precedence relations which must be taken into account in building the schedule.

A necessary and sufficient condition for the existence of a valid schedule derived from the transformation to an instance of the process model is given as a utilization factor. The utilization factor is the computation time for each process divided by its period summed over all the processes. The necessary and sufficient condition for the existence of a valid schedule in single processor systems is a utilization factor that is less than or equal to one.

Once the elements of the graph are transformed into elements of the process model, an earliest deadline first algorithm or earliest deadline - predecessor priority algorithm can be used to build the schedule. The paper concludes with the theorem that a valid schedule for a graph model exists only if there is a valid schedule for the process model.

Work is also being done in the scheduling of dataflow systems by Bic [Ref. 13]. He describes a model for efficient execution of data flow. His goal is to reduce run time overhead inefficiencies by breaking a data flow program into sequences of instructions that must be executed sequentially due to their data dependencies. The expected benefit of the algorithm is the high degree of parallelism during execution. This work is not directly applicable to the static scheduler design since the static scheduler for PSDL is being designed to combine both data flow and control flow elements.

Mok presents a methodology to automate the synthesis of code for time critical applications which takes into account resource (hardware) requirements [Ref. 14]. The general strategy involves representing hard real-time design specifications as instances of a formal graph-based computational model. Resource allocation analysis is performed on each problem instance to arrive at an implementation which meets the specified critical timing constraints.

The graph based model is an ordered pair (G,T) made up of a communication graph (G) and a set of timing constraints (T). The communication graph is a directed

graph made up of nodes and edges. The set of timing constraints consists of a task graph (C), period (p), and deadline (d). The task graph is an acyclic directed graph which defines the precedence relation of the computational events that must occur in order to satisfy a timing constraint. Timing constraints can either be periodic or asynchronous.

Implementing an instance of this model is done by mapping each periodic/asynchronous timing constraint into a periodic/asynchronous process. The process is any topological sort of the operations in the task graph. The static schedule resulting from this model is a finite string of functional elements and idle times. This idea describes the PSDL static schedule which is, at each level of decomposition, a string of operators and idle times.

Mok describes three algorithms for decomposing the computational requirements of real-time systems into a set of processes with critical timing constraints [Ref. 15]. This is a typical first step in automating the synthesis of real-time systems and the algorithms are:

1. Decomposition by Critical Timing Constraints (CTC)
2. Decomposition by Centralizing Concurrency Control (CCC)
3. Decomposition by Distributing Concurrency Control (DCC)

a. Decomposition by Critical Timing Constraints

In decomposition by critical timing constraints, a process composed of a sequence of function calls to programs implementing functional elements of a timing constraint is created to perform the computation associated with a timing constraint. A special process called a monitor is created to enforce mutual exclusion on the execution of a function called by two or more processes. While this decomposition is the most straightforward way to decompose the design, it is not the most efficient since some computations may be duplicated in two or more processes.

b. Decomposition by Centralizing Concurrency Control

In decomposition by centralizing concurrency control, periodic timing constraints that are compatible with one another are grouped together into an equivalence class. Two timing constraints are defined to be compatible if period 1 divides evenly into period 2 or period 2 divides evenly into period 1 (one deadline is an exact multiple of the other), deadline 1 equals deadline 2, and task graphs 1 and 2 have some nodes in common. A periodic process is then created for each equivalence class. In this way, redundant computation is avoided and since concurrency control is centralized, processes

tend to be independent of each other. One drawback to this algorithm, however, is that it may not yield the shortest program possible. This type of decomposition is useful in developing the harmonic block concept for the PSDL static scheduler.

c. Decomposition by Distributing Concurrency Control

In decomposition by distributing concurrency control, a periodic process will be created for each functional element in the communication graph. Since a functional element can occur in two or more task graphs, the periodic process which is created will have a period equal to the smallest period among the periodic timing constraints in which the functional element occurs. In this way, the computation is decomposed into as many processes as possible and results in increased parallelism in the design. However, the resulting designs tend to be more difficult to understand and modify if necessary.

Once a real-time system is decomposed into processes, these processes must be scheduled. Mok examines the real-time scheduling problems of three process models and discusses the implication of his results on the design of real-time programming languages [Ref. 16].

Real-time scheduling deals with the problem of continuously meeting sporadic and periodic timing constraints. The three models discussed are:

1. The Independent Processes Model
2. The Deterministic Rendezvous Model
3. The Kernelized Monitor Model.

All of these models deal with the single processor case.

a. The Independent Processes Model

The independent processes model assumes that processes are preemptible and in that situation, the earliest deadline first scheduling algorithm is optimal. However, when preemption is not allowed, which is the case in the PSDL static scheduler, earliest deadline first is no longer optimal.

b. The Deterministic Rendezvous Model

In the deterministic rendezvous model, a rendezvous primitive is used for synchronizing two processes and establishing precedence constraints. For scheduling purposes, the rendezvous is assumed to take zero time. The earliest dealine first algorithm which is modified to run the ready process with the nearest deadline and which is not blocked by a rendezvous primitive is not optimal. This problem is easily fixed by

adopting a technique for revising deadlines to eliminate precedence constraints which then allows the use of the earliest deadline first algorithm.

The technique used to revise the deadlines first sorts the scheduling blocks contained in each process in reverse topological order. Then the deadline of a scheduling block is moved up if it must precede another scheduling block which has a nearer deadline but is not yet ready to run. The revised deadlines are used to update the current deadlines at run-time.

The feasibility of the process model is not affected by dynamically updating the process deadlines as described above if all of the processes in the model are periodic.

c. The Kernelized Monitor Model

The third model is the kernelized monitor model. The operating system enforces mutual exclusion by allocating processor time only in uninterruptible quanta of any size which is chosen to be bigger than the largest monitor. A monitor is a special type of process that performs some service for ordinary processes on demand. The only difference between this model and the deterministic rendezvous model for scheduling purposes is that a process may be interrupted only after it has been allocated an integral number of quanta.

The implications of this work on the design of real-time languages were evaluated against the criterion of a language construct's impact on software automation. By this criterion, the use of semaphores is undesirable, there is substantial benefit in making the enforcement of interprocess synchronization and mutual exclusion syntactically distinct, and there may not be any easy way to deduce whether a control construct is being used to enforce a synchronization or a mutual exclusion constraint.

A scheduling algorithm that guarantees the response times for a fixed set of tasks run on a single dedicated processor in a hard real-time environment is described by Leinbaugh in [Ref. 17]. A task's guaranteed response time is defined as the maximum time from the successful start of the task to it's completion. Leinbaugh's model allows for input, output requirements and competition among tasks for the same devices or data in addition to central processor requirements for each task. A task is defined as a series of segments which are run one at a time in order. Guaranteed response times are computed from knowing the maximum processor time needed for each segment, the maximum operating time of each device, the symbolic resources needed by each segment and the placement of device requests within each segment and the placement of segments

within each task. Resource segments have priority over other segments and requests for devices are run on a first-come-first-served basis.

Curtis Abbot presents an intervention scheduling model for programming real-time systems to run on single processor and shared memory multiprocessor systems [Ref. 18]. Intervention schedules are characterized by a sequence of events. Events are enabled when triggers fire and once processing of an event starts, it runs to completion (nonpreemptive processing). In Abbot's model, an intervention schedule can be thought of as a list of expressions with wait conditions interspersed among them. Each wait condition names a trigger. Intervention schedules, therefore, are ordinarily in a waiting state as opposed to processes that are ordinarily running and occassionally waiting to synchronize with cooperating processes.

Intervention schedules organize programs such that timing is separated from computation. This separation is accomplished partly by the fact that no data accompanies a trigger. For this reason, this research is of little help in the design of the static scheduler for PSDL implementation where data can be the trigger.

Each of the above scheduling algorithms have tried to capture timing constraints in a temporal sense. Peter Ladkin discusses how time dependencies may be more appropriately specified using the interval calculus rather than temporal logic [Ref. 19]. He contends that without considerable training in logic, it is difficult for many programmers to write correct specifications in using temporal logic and that temporal logic is only suitable for sequencing and concurrency control. By contrast, Ladkin argues that the interval calculus is a more natural means of specifying concurrent systems behavior and is more suitable for specification of real-time systems. The high level interval calculus formulation is then refined by an automatic process into a lower level specification. Despite this contention, timing constraints in PSDL will be handled temporally.

2. Software Safety

When designing hard real-time systems, it is important to determine whether timing constraints are being satisfied since the failure to meet one deadline could result in catastrophic loss. Safety assertions, which can be thought of as additional constraints on the system, can be used to perform a safety analysis. Formalizing the safety analysis of timing properties in real-time systems is the objective of Jahanian and Mok [Ref. 20]. The properties of real-time systems are described in an event-action model which is then translated into in Real-Time Logic (RTL) formulas. Safety assertions are treated as additional constraints on the system and are also translated into RTL. RTL

formulas are translated into predicates of Presburger Arithmetic and existing functions can then be used to determine if a given safety assertion is a theorem derivable from the system's specification. The ultimate goal is the creation of a practical tool for software engineers to use in analyzing real-time systems.

For a system to be considered safe, the safety assertion must be a theorem which is derivable from the system specification. The system is considered inherently unsafe if the safety assertion is unsatisfiable with respect to the specification. Finally, if the negation of the safety assertion is satisfiable under certain conditions, additional constraints must be imposed on the system to ensure its safety. Jahanian and Mok, [Ref. 21], describe a three-part graph-theoretic procedure for safety analysis for certain timing properties that are expressible in a subset of Real Time Logic formulas. The first part of the approach constructs a graph that represents the system specification and the negation of the safety assertion. The second part of the analysis uses a node removal procedure to detect positive cycles in the graph. The third part determines the consistency of the safety assertion with respect to the system specification. The safety assertion is considered consistent with the specification if an RTL formula consisting of the specification in conjunction with the negation of the safety assertion is unsatisfiable. This version of the safety analyzer is being implemented as part of a design environment for real-time software known as SARTOR.

The initial conceptual design presented in this thesis is for a prototype of the PSDL static scheduler. Therefore, while the importance of safety analysis is recognized, it is not dealt with in any detail in this design.

3. Execution Monitoring

It is not enough in most cases to simply build a valid static schedule, execute the prototype, and decide whether the system works or fails. Since a valid static schedule is a necessary but not sufficient condition for successful execution, prototype execution needs to be monitored to determine where problems in timing may occur and to collect run-time data. Bernhard Plattner, [Ref. 22], proposes an execution monitoring process that is conducted in real-time and on a symbolic level, where the monitor and operator communicate in terms of the source code that the monitored or target process is written in. The operator describes what is to be monitored in a monitoring statement which consists of a predicate and an action. The action is executed only when the predicate equates to a boolean value of true. In this way, information about a process can be

recorded. To prevent the monitoring system from affecting the run-time execution of the target process, the monitor and the target process do not share the same processor.

In the PSDL execution support system, the task of collecting run-time data is delegated to the dynamic scheduler and so will not be discussed in this thesis.

C. SUMMARY

This brief look at the ongoing research efforts in hard real-time system development underscores two important facts. First, the design of hard real-time systems and their associated scheduling considerations are receiving a great deal of attention. The importance of this again is reduction of time, effort, and money in the software design process of systems whose error-free performance is not only desirable but mandatory and the unique design problems for real-time systems are being acknowledged and dealt with. Second, this survey highlights the fact that though many research efforts are in progress, the Computer Aided Prototyping System and Prototype System Description Language are not being duplicated by other research efforts. As a result, much of the information presented in this survey chapter, though informative, is not directly applicable to the design of the PSDL static scheduler. The conceptual design for the PSDL static scheduler is presented in Chapter III.

III. CONCEPTUAL DESIGN FOR THE STATIC SCHEDULER

As discussed briefly in Chapter II, the static scheduler is responsible for scheduling time critical operators in such a way that all timing constraints as well as precedence relations are met. Figure 7 on page 23 is the first level data flow diagram for the static scheduler as conceptualized by this author. A PSDL source file, generated by the system designer, contains the PSDL operator specifications and implementations for the prototype. In Read_PSDL, the static scheduler reads this file and collects operator names and timing information. The resulting file is then run through a Text_file_preprocessor where operators are separated into time critical and non-time critical files. The non-time critical operators are sent to the dynamic scheduler. The static scheduler retains the time critical operators for itself. There are several conditions between the timing constraints that must be true in order for the prototype to run properly. Before any operators will be scheduled, a series of simple validity checks will be conducted to ensure that the proper relationships hold between the time critical operators.

In order to maintain the precedence relationships among operators in the final static schedule, the operators must be sorted topologically. This is done in Topological_sort.

Next in the data flow diagram is Build_harmonic_blocks. A harmonic block is a set of operators such that each operator in the set has a period that is an exact multiple of the base period and at least one of the operators has a period equal to the base period [Ref. 7: p. 7]. The last requirement may be relaxed in the single processor case since there is only one harmonic block. Sporadic operators will be given an equivalent period prior to constructing the harmonic block(s). It makes no difference whether the operators are first sorted in topological order or organized into harmonic blocks since both activities yield separate schedules.

Finally, Schedule_operators combines the separate harmonic block and topological schedules into one static schedule that meets both timing constraints and precedence relationships. In the multiprocessor case, each harmonic block is a separate scheduling problem handled by separate processors. In the single processor case, only one harmonic block is constructed. The final static schedule will be a finite string of operators meeting worst case execution times and occasionally separated by idle time slots resulting from operator periodicity constraints. This will become clearer in Section E. The dynamic scheduler will schedule non-time critical operators into these idle time slots as

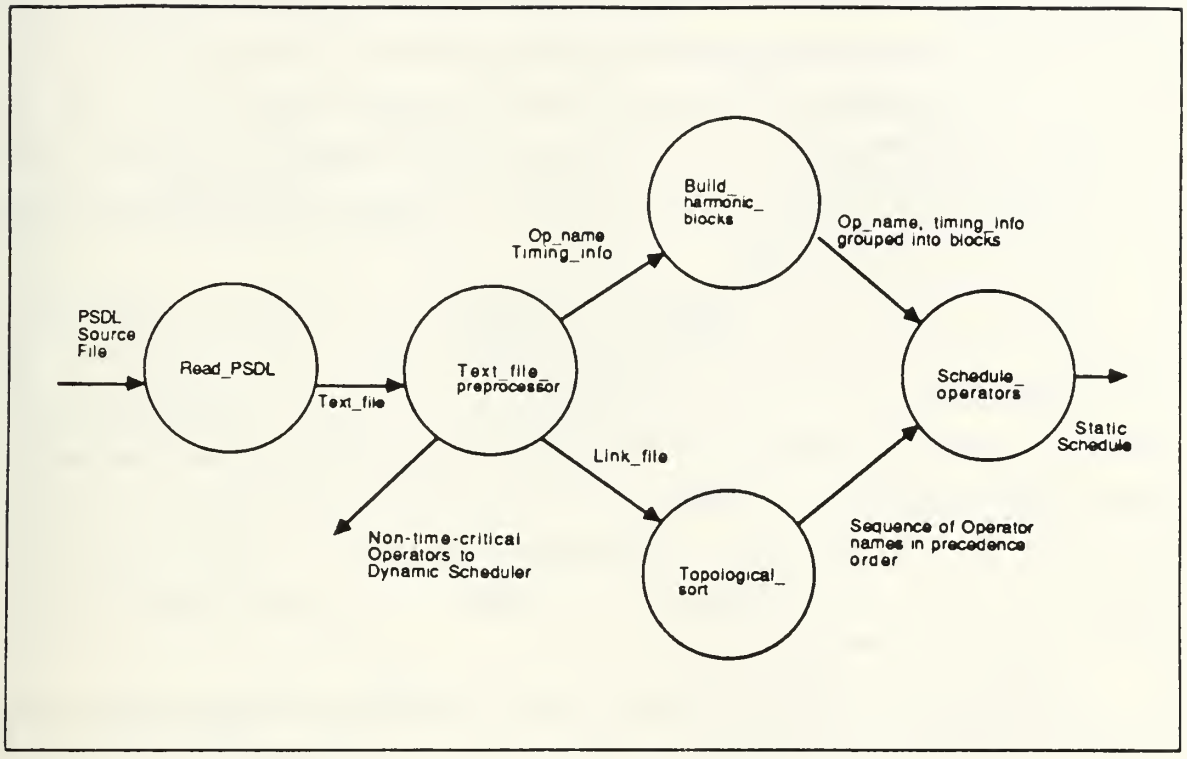


Figure 7. Static Scheduler, 1st Level Data Flow Diagram

well as any time remaining if a time critical operator completes execution prior to its worst case execution time.

The remainder of this Chapter is organized into sections corresponding to each of the bubbles in Figure 7. Where applicable, the algorithm for each module of the static scheduler is presented followed by an example of its execution.

A. READ_PSDL

As it is being designed, the static scheduler will obtain PSDL operator timing and precedence information directly from the PSDL prototype source file. Specifically, the static scheduler is looking for an operator's maximum execution time, minimum calling period, maximum response time, and period. The maximum execution time (MET) is an upper bound on the length of time between the instant when a module begins execution and the instant when it completes [Ref. 6: p. 23]. The MET applies to both periodic and sporadic operators. The maximum response time (MRT) for a sporadic operator is an upper bound on the time between the arrival of a new data value and the time when the last value is put into the output streams of the operator in response to

the arrival of the new data value. The MRT for a periodic operator is an upper bound on the time between the beginning of a period and the time when the last value is put into the output streams of that operator during that period. [Ref. 6: pp. 23-24] The minimum calling period (MCP) is a constraint on the environment of a sporadic operator consisting of a lower bound between the arrival of one set of inputs and the arrival of the next set [Ref. 6: p. 24].

There are three common definitions of a period. The first is that an operator is scheduled to fire at exact time intervals equal to the period. For a period of 10, this means that exactly every 10 time units the operator would fire. The second definition of a period is that the operator must be scheduled to fire within the stated time interval. Again, for a period of 10, the operator must fire sometime between time t = 0 and 10, 10 and 20, 20 and 30, and so forth. The third definition is that an operator must fire and complete execution within the specified time interval. This third definition is the one used in designing the static scheduler.

In addition to the timing constraints, the static scheduler must also obtain the precedence relationships between the operators. Precedence relationships are shown in PSDL as directed graphs. A directed graph is a set of nodes and edges with arrows indicating the direction of dataflow on the edges. For purposes of the topological sort, the static scheduler will have to differentiate between operators that are functions and those that are state machines. When a function fires, outputs depend only on the set of current input values. Functions are represented as acyclic digraphs. When a state machine fires, its outputs depend on the set of input values and a finite number of internal state variables. State machines are shown as cyclic digraphs. Figure 8 on page 25 shows this distinction graphically. As discussed in Section C, the topological sort algorithm applies to acyclic digraphs only.

Now that we know what information we are looking for, we must determine where it can be found within the PSDL source file. Appendix A contains the PSDL grammar. As defined in the grammar, PSDL operators have two major parts, specification and implementation. The MET, MCP, and MRT can be found in the specification part. These timing constraints are either stated explicitly or, in the case of the MCP, are inherited from a higher level operator. In addition to the timing characteristics, the operator specification contains attributes describing the form of the interface and formal and informal descriptions of the observable behavior of the operator. The "States" attribute

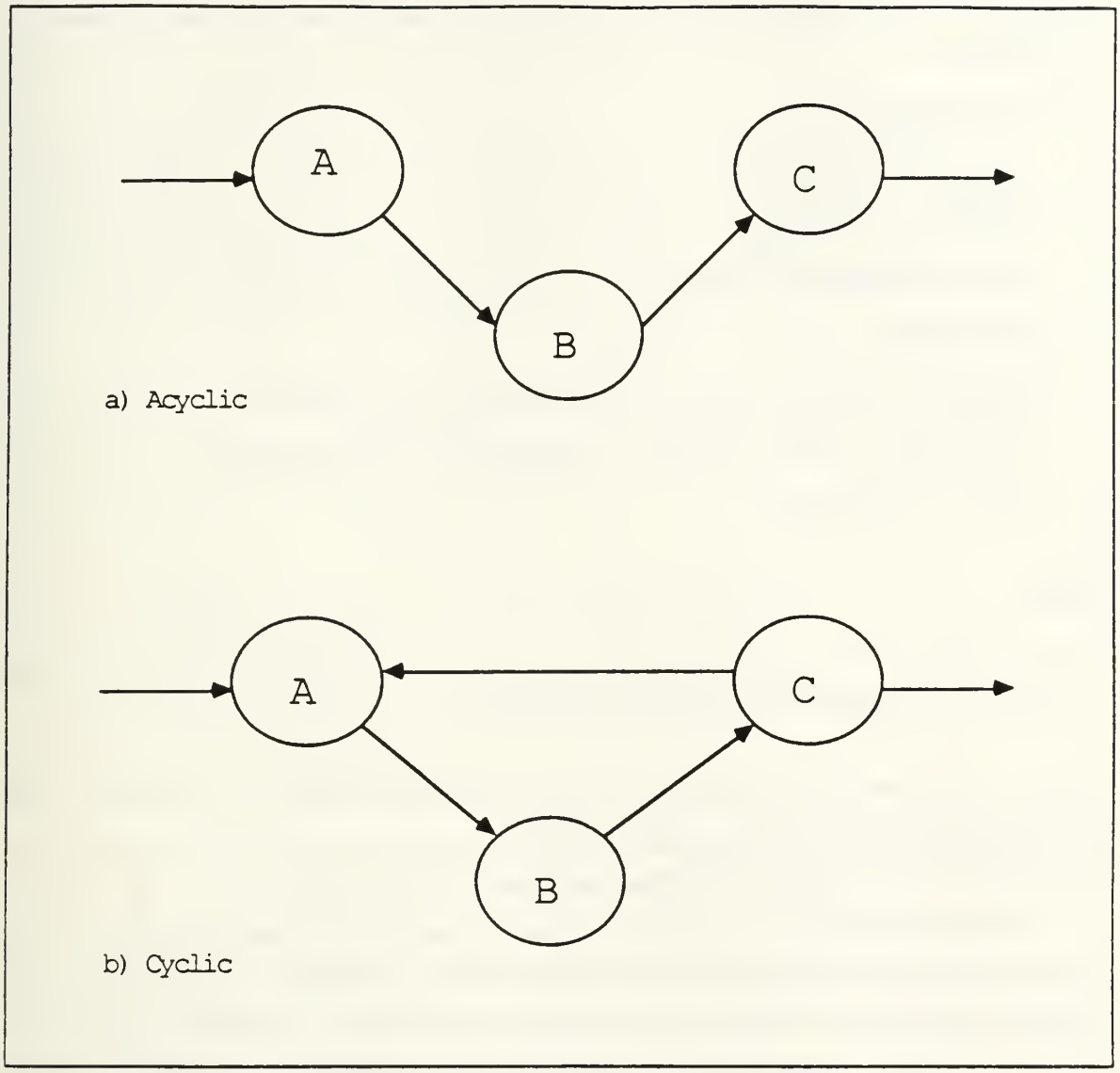


Figure 8. Example of Acyclic and Cyclic Digraphs

identifies an operator as a state machine and is one of the attributes found in the specification part. Figure 9 on page 26 shows an example of an operator specification.

The precedence relationships and the period are found in the operator implementation part. A composite operator (one that can be realized as lower level operators) contains a graph depicting the relationships among the lower level operators. While the user sees a directed graph on the terminal, the system designer has used PSDL statements called link statements to indicate the direction of data flow. An acyclic directed

```
OPERATOR B
SPECIFICATION
  INPUT b : integer
  OUTPUT c : integer
  MAXIMUM EXECUTION TIME 2 ms
DESCRIPTION
{ Operator B takes as input data stream b and outputs
  a response on data stream c.  Operator B has a maximum
  of 2 ms to do this.
}
END
```

Figure 9. Example of an Operator Specification

graph with its corresponding link statements is shown in Figure 10 on page 27. The times specified in the link statements are the
maximum execution times of the operators.

The first link statement (a.EXT → A) and the last link statement (d.C:2ms → EXT) both contain a pseudo-operator named EXT. EXT (for external). This is a convention used by this author to identify those operators in the graph that have data coming from or going to some operator external to a particular decomposition. Even though EXT is not found in the PSDL grammar, it is required by the static scheduler in order to execute the topological sort algorithm. Whenever EXT is seen in a link statement, it can be interpreted to mean that the operator is either the first or the last in the graph. For the first operator in the graph this means that it has no incoming edges. This will be significant when executing the topological sort algorithm. For the last operator in the graph, whether or not it has an outgoing edge is immaterial.

A link statement describes the relationship between two operators by indicating the direction of dataflow between them. A link statement is to be read as data_stream.from_operator → to_operator. The first link statement in Figure 10 on page 27 is therefore read as data_stream "a" connects an "external operator" to operator

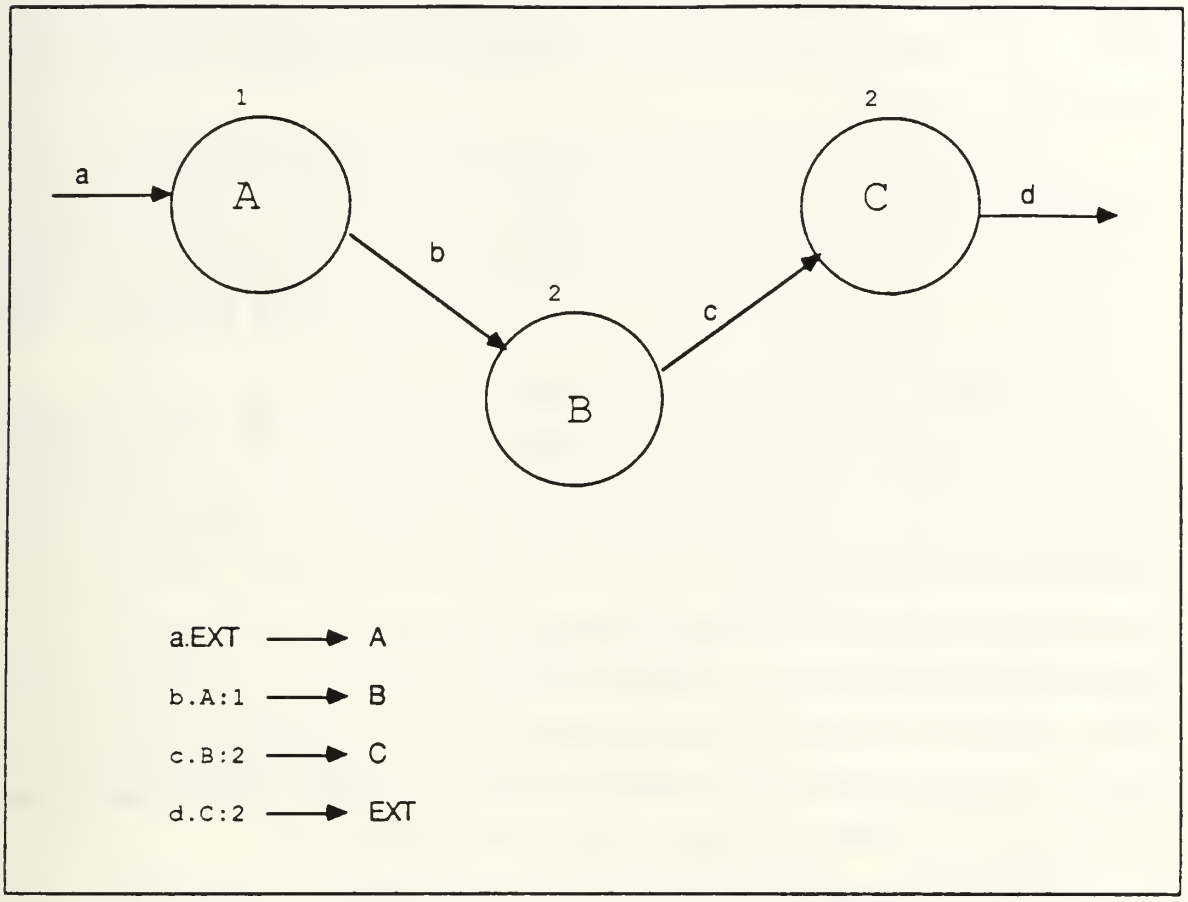


Figure 10. Link Statements Associated With a Directed Graph

A. In other words, operator A is the first operator in the graph and has no incoming edges. The second link statement links operators A and B with data_stream b. The order of the operators in a link statement is always in the direction of dataflow.

A periodic operator's period is found in the implementation section under "control constraints." If a period is not explicitly stated, it is inherited from a higher level operator. If the maximum response time was not explicitly stated in the operator specification part, it may be found in the implementation part as "finish_within."

The static scheduler will process the PSDL source file using an attribute grammar (AG) based tool. This tool allows the user to define what attributes of the PSDL file are important. All undefined attributes are ignored by the processor. The PSDL attributes that will be defined for use by the static scheduler are "operator," id, operator "specification," "states," "maximum execution time," "minimum calling period," "maximum response time," time, unit, operator "implementation," "graph," link statements.

control_constraints, "period," "finish_within," psdl_impl, constraint, op_id, and attribute. The operator names and corresponding timing and precedence information will be stored in a text file.

It is assumed by this author that the PSDL source file will be written hierarchically starting at the highest level of abstraction. The static scheduler will read the file from beginning to end and should, therefore, collect data from the top down as opposed to bottom up. In other words, Operator A's timing information will be recorded before Operators A1, A2, and A3.

An example of a prototype written in PSDL is contained in Appendix B. This is a prototype for a real-time system for the treatment of brain tumors. Hyperthermia induced microwaves are applied directly to the tumors, effectively killing them. A computerized control system is used to adjust power output automatically to maintain the proper therapeutic temperature.

As is typical with real-time systems, computer software controls the operation of the whole system which is made up of four subsystems, a computer system, an operator panel, a microwave generator, and a temperature sensor. The PSDL example in Appendix B is the prototype for the computer software subsystem. The remaining three subsystems will be simulated in order to demonstrate the prototype.

The first operator in the prototype is brain_tumor_treatment_system. It is a state machine with the states variable being temperature which is initially 37°. This operator is a composite operator and is decomposed into operator hyperthermia_system and operator simulated_patient. Each of these operators is periodic with a period of 200. Although additional information is contained in the PSDL specification and implementation, it is not pertinent to the operation of the static scheduler.

At the second level in the prototype hierarchy, operator hyperthermia_system is described in more detail. It has a maximum execution time of 100 ms and a maximum response time of 300 ms. It is also a composite operator and can be decomposed into operators start_up, maintain, and safety_control. These last three operators are atomic and are implemented in Ada rather than decomposed further. Each of these operators has a maximum execution time which the static scheduler will obtain from their specification parts.

This is a typical application of PSDL in building a prototype. The objective is to build a bare bones prototype of the critical subsystems of a larger system in order to demonstrate its feasibility. In the hyperthermia system, as in most embedded control

systems, it would be too costly and dangerous to build the complete system and demonstrate it in its real world environment.

B. TEXT_FILE_PREPROCESSOR

The second level data flow diagram for the text_file_preprocessor, shown in Figure 11 on page 30, indicates two basic activities, locating time critical operators and performing simple validity checks on timing information.

1. Separate_critical_operators

The text file created by reading a PSDL source file contains the names of all of the operators in the prototype as well as any timing properties that may be associated with them. The static scheduler must separate out the time critical operators from the non-time critical operators in order to pass the non-time critical operators to the dynamic scheduler. This is done by reading the text file and flagging or otherwise identifying the operators that do not have any timing properties. The static scheduler assumes that an operator is time critical if it has associated with it at least one of the following:

Maximum execution time,
Minimum calling period,
Maximum response time, and
Period.

A separate text file is created which contains only the non-time critical operators and they are deleted from the original text file. This results in the creation of two text files, one containing non-time critical operators and the other containing the time critical operators. The static scheduler does no further processing of the non-time critical operators. The remainder of this chapter assumes all operators are time critical.

The Text_file_preprocessor will also separate the link statements associated with each operator into another separate file. In addition, Link statements associated with a state_machine will be deleted from the file to prevent the occurrence of cyclic digraphs. This is accomplished by identifying the states variable(s) for each operator. States variables always appear in the graph as data_streams. Because data_streams always come first in the link statement, it is a simple matter to compare the states variable with the first position in each link statement and delete those that match. The resulting file should consist of a set of link statements which produce an acyclic directed graph. This is the file that will be sorted topologically to create a schedule showing the precedence relationships between the operators.

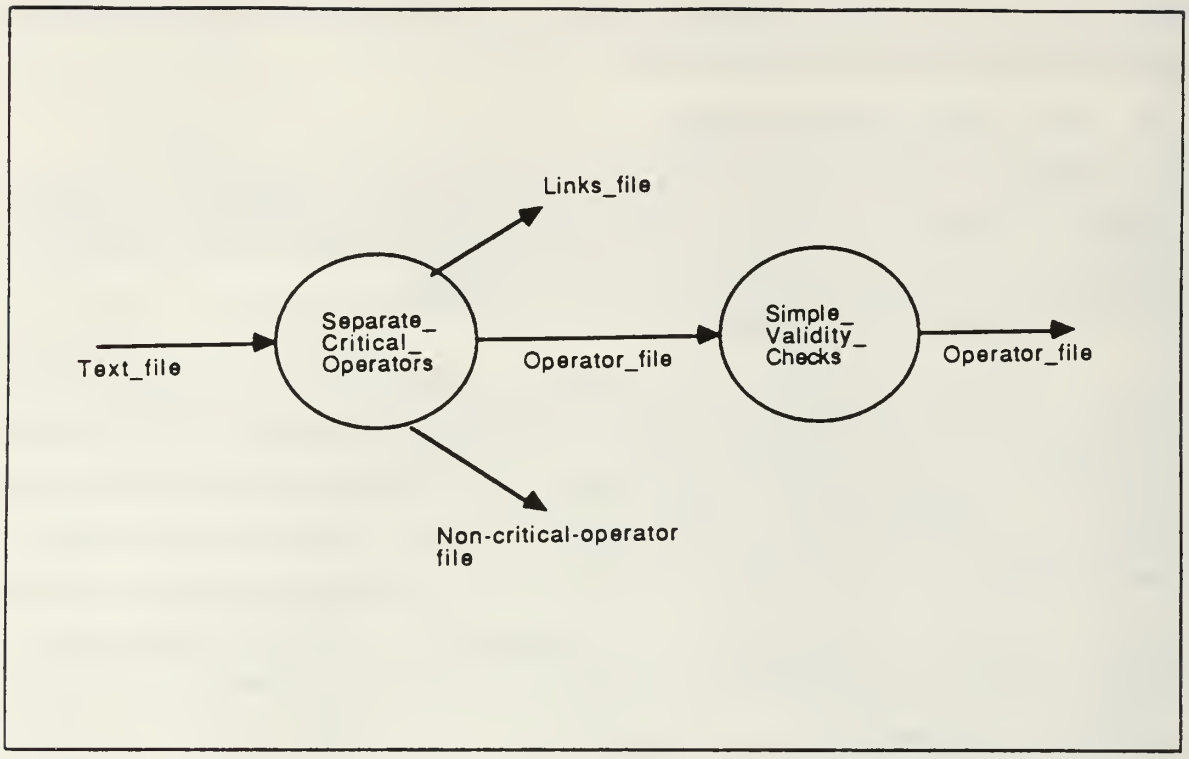


Figure 11. Text_file_preprocessor, 2nd Level Data Flow Diagram

2. Simple_validity_checks

There are certain relationships which must hold between some of the timing constraints in order for the prototype to function properly. First, the maximum execution time for an operator must be greater than or equal to its actual execution time. Since the static schedule is built on worst case execution times, it follows that if actual execution time is greater than the scheduled execution time the system will fail.

Second, the maximum execution time for an operator must be less than its maximum response time. Since the maximum response time is defined as the upper bound on when an operator puts it last value into an output stream, it is obvious that the execution time must be sufficiently less than the response time to accomplish processing prior to the time when the output is required.

Third, an operator's period must be greater than its maximum execution time. If the period were less than the maximum execution time, an operator would be scheduled to fire again before it had completed execution from its previous firing. This would cause the next firing to be delayed, thereby delaying the remainder of the static schedule. This in all probability, will cause the system to fail.

These first three timing relationships apply to an individual operator. There is an additional relationship that must hold betweeen the operators. The sum of the maximum execution times of atomic or lower level operators must be equal to or less than the maximum execution time of the composite operators at the next higher level in the hierarchy. For example, assume that composite operator X has a maximum execution time of 100 milliseconds. If operator X is decomposed into operators X1, X2, and X3, the sum of the maximum execution times of these three operators cannot exceed 100 milliseconds.

This is not a complete listing of all the validity checks that are possible but these should be sufficient for this prototype to weed out the obvious errors. In the ideal situation, the static scheduler would be smart enough to make appropriate corrections or substitutions to maintain these relationships without causing the program to cease execution. For this prototype, however, if an inconsistency is discovered during the execution of the validity checks, an exception will be raised notifying the software designer of the problem and execution will cease.

C. TOPOLOGICAL_SORT

The second level data flow diagram for Topological_sort is shown in Figure 12 on page 32. As shown, there are three general sections to this algorithm. It is a simple algorithm that essentially finds that operator which must precede all others in a set, concatenates that operator to a sequence of operators, and then deletes that operator and all its edges from the set. This cycle is repeated until all operators have been deleted from the set and it is empty. The final sequence should contain all operator names, in order, by precedence.

1. Find_first_operator

The operator that must precede all others in the set can be identified easily from the set of link statements because that operator will not have any incoming edges. It will have either the word EXT on the left hand side of the arrow in the link statement or it will be an operator that only appears on the left hand side of the arrow. Since the link statements are always written in from-to form, an operator that is named on both the left and right hand sides of an arrow in two or more link statements always has at least one incoming edge.

In situations where more than one operator is identified as being first, the static scheduler will arbitrarily choose one. It should not make a difference in the final schedule since operators so identified have no precedence relation between them.

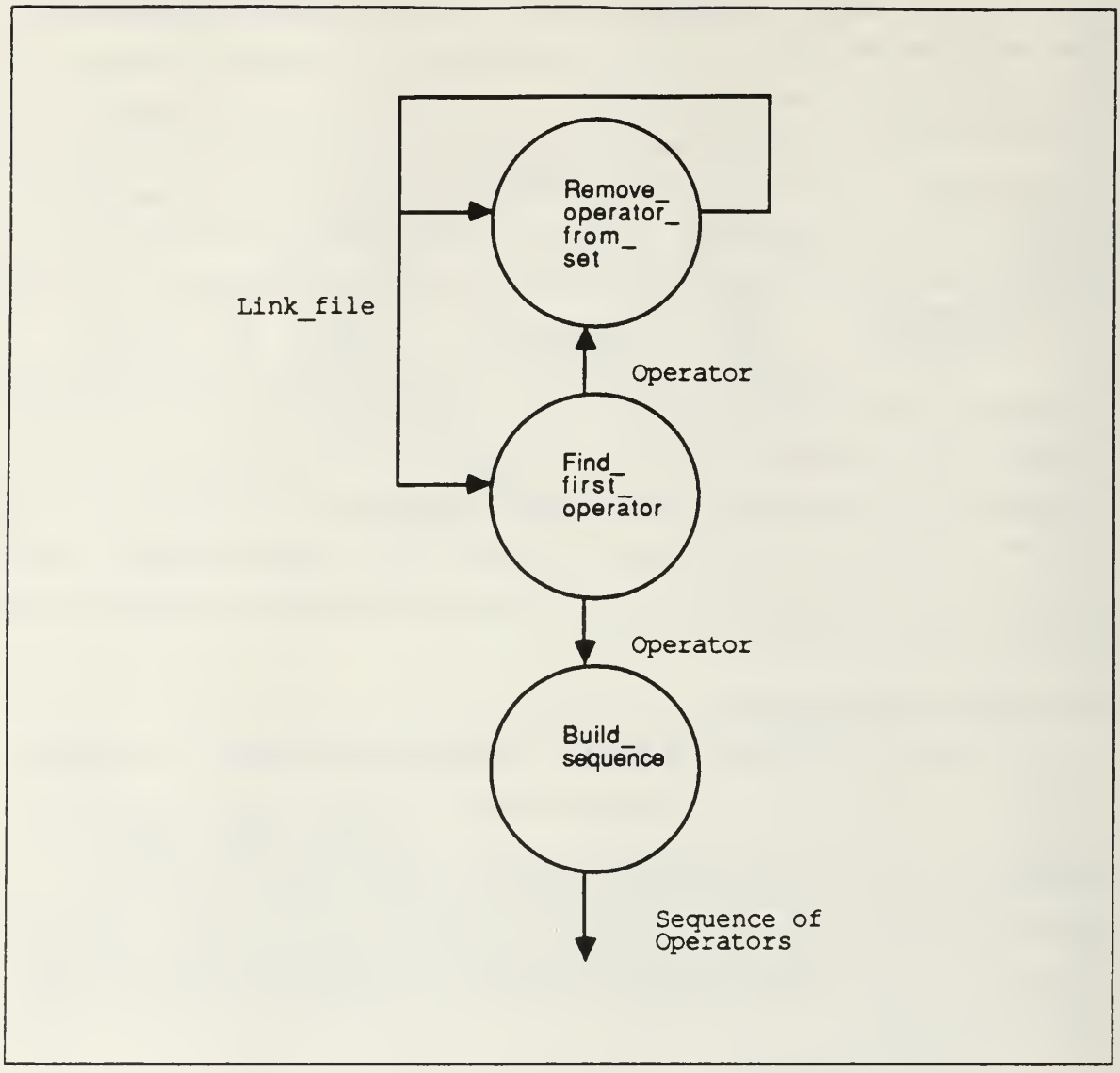


Figure 12. Topological_sort, 2nd Level Data Flow Diagram

2. Build_sequence

Next, the operator that has been identified as the first operator is concatenated to a sequence. Initially, the sequence is empty. Eventually it will contain the names of all operators in the decomposition. If more than one operator is identified as having no inconing edges, each operator is concatenated to the sequence arbitrarily.

3. Remove_operator_from_set

Finally, all the link statements associated with the operator(s) just concatenated to the sequence will be deleted from the set. This effectively removes the operator and

all of its edges. The algorithm will repeat these steps, searching the new, smaller set for that operator which has no incoming edges. Execution of this algorithm continues until the set of link statements is empty.

The complete algorithm is listed below:

1. Create the sequence. Initially it will be empty.
2. While the set of link statements is not empty, search the set for operators that do not have any incoming edges. (EXT to the left of the arrow or operator name that appears to the left of an arrow, never on the right).
3. If more than one operator has no incoming edges, arbitrarily select the operator that was located first.
4. Concatenate the operator name to the sequence.
5. Delete all the link statements in the set that contain the operator name either to the left or the right of the arrow.
6. Repeat steps 2 through 6 until the set is empty.

The directed graph shown in Figure 10 on page 27 will be used to proved a simple example of the topological sort algorithm. This graph is an acyclic graph. However, had it been a state machine represented as a cyclic graph, link statements containing the state variable will aready have been deleted prior to the execution of the sort algorithm. The example is outlined below:

Step 1. Precedence_list : sequence. Initially it is empty ([ ]).

Step 2. Statements : set.

Initially it looks like { a.EXT → A, b.A:1 → B, c.B:2 → C, d.C:2 → EXT } . Since Statements is not empty, search for operators that do not have any incoming edges (either EXT to the left of the arrow or an operator to the left of the arrow but not on the right).

Since operator A had data coming from EXT, it qualifies under this rule. Operators B and C do not qualify since both appear on either side of the arrow.

Step 3. Not applicable.

Step 4. Concatenate A to Precedence_list. Precedence_list looks like [A] .

Step 5. Delete all link statements in Statements that contain A. Statements looks like { c.B:2 → C, d.C:2 → EXT } .

Step 6. Repeat steps 2 - 6.

Step 2. Statements is not empty. B is the only operator that does not appear on both sides of a link statement arrow.

Step 3. Not applicable.

Step 4. Concatenate B to Precedence_list. Precedence_list looks like [A, B] .

Step 5. Delete all link statements in Statements that contain B. Statements looks like { d.C:2 → EXT } .

Step 6. Repeat Steps 2 - 6.

Step 2. Statements is not empty, operator C is the only remaining operator.

Step 3. Not applicable.

Step 4. Concatenate C to Precedence_list. Precedence_list looks like [A, B, C] .

Step 5. Delete all link statments containing C from Statements. Statements looks like { } .

Step 6. Repeat Steps 2 - 6.

Step 2. Statements is empty, execution is finished.

The final precedence list is [A, B, C] .

D. BUILD_HARMONIC_BLOCKS

The second level data flow diagram for Build_harmonic_blocks is shown in Figure 13. All of the operators in a harmonic block are required to have periods that are exact multiples of the base period. Therefore, a sporadic operator must be assigned a period which is known as its equivalent period. Once equivalent periods are assigned, all operators are treated as periodic operators. To simplify the Build_harmonic_block algorithm, the operators are sorted by period in ascending order. Once this is accomplished, individual operators can be separated into the appropriate harmonic block based on the definition given at the beginning of this chapter. Finally the block length is calculated.

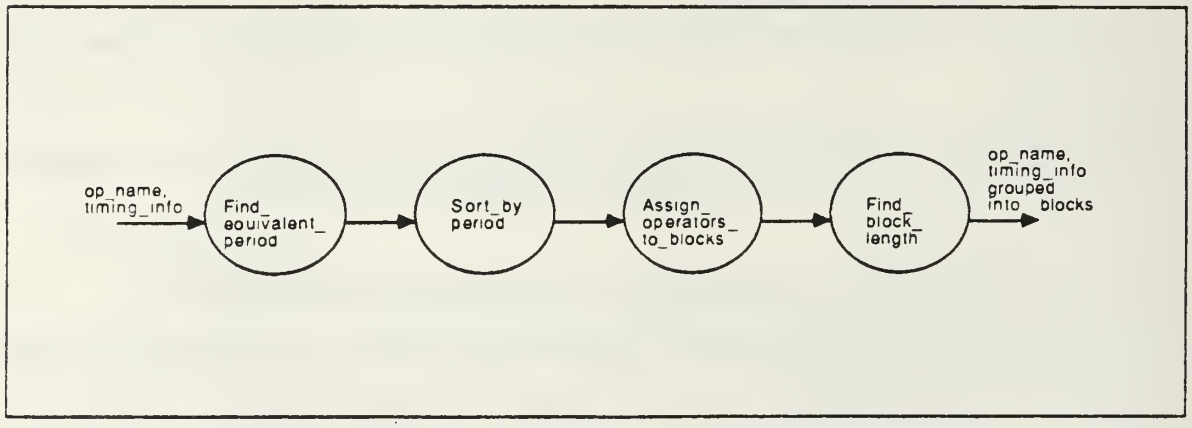


Figure 13. Build_harmonic_blocks, 2nd Level Data Flow Diagram

1. **Find_equivalent_period**

A sporadic operator's equivalent period is determined by the formula

$$P = MIN(MCP, MRT - MET).$$

An equivalent period derived in this way has a deadline equal to the MET. As a result, these operators must be scheduled to start at the beginning of the period. Since period - deadline = slack time, P - MET ≥ 0 for the sporadic operator. A constant phase shift is allowable, however. In addition, the period must be greater than or equal to the execution time of the operator so that it can meet its timing constraints. [Ref. 7: p. 8].

There is some debate over whether the equivalent period must be greater than, equal to, or less than the minimum calling period (MCP). The MCP specifies a minimum amount of time that must pass between the arrival of two successive inputs. This prevents the operator from being continuously called on to perform its function. If the period is less than the MCP, the operator will be scheduled to fire before new data values are allowed on the input stream. Depending on the type of data stream employed by the operator, this may cause the prototype to fail. Also, the prototype could fail because, as stated above, the operator must be scheduled to fire at the beginning of a period in order to meet its deadline. However, both of these conditions do not preclude a valid static schedule from being constructed so there is no reason to require that P be greater than the MCP for this type of static scheduler.

The algorithm for determining the equivalent period for a sporadic operator is for each sporadic operator in the Text_file do:

1. Compute the equivalent period P using the formula P = MIN(MCP, MRT - MET).
2. Compare P and MET. P must be $\geq$ MET. If it is not, make it equal to MET.

For example, if operator X has an MCP of 2, an MRT of 10, and an MET of 5, following Step 1 above, P = MIN(2, 5) or P = 2. Since this is less than the MET of 5, P will have to be changed to 5.

At this point, a test can be done on the operator periods to assess the feasibility of building a valid static schedule. Specifically, an operator's maximum execution time divided by its period summed over all operators must be less than or equal to the number of processors available in order for a valid schedule to be possible. This is represented mathematically as

$$\Sigma \left(\text{MET}(i) / \text{Period}(i) \right) \leq \text{number of processors.}$$

If this relationship does not hold, an exception will be generated and the designer will be advised of the nature of the problem. As with the previous simple validity checks, execution of the static scheduler will be suspended until the appropriate corrections are made.

2. Sort_by_period

This requires a simple sort procedure. The operators will be sorted by period from smallest to largest. The specified or inherited period will be used for periodic operators and the equivalent period will be used for sporadic operators. From now on, all operators are treated as periodic. The operators are sorted by period to make the determination of harmonic block base periods easier.

3. Assign_operators_to_blocks

Finally, the operators are ready to be assigned to harmonic blocks. The algorithm is different for single and multiprocessor environments. Each will be described separately.

a. Single processor

In the case where there is only one processor, N = 1, all operators will belong to one harmonic block. The requirement that one of the operator periods in the block be equal to base period is relaxed. Instead, the base period is the greatest common divisor (GCD) of all the operators in the set. Z is defined as the GCD(X,Y) if and only if X Mod Z = 0 and Y Mod Z = 0 and (X Mod W = 0 and Y Mod W = 0) implies $W \leq Z$. Mod refers to the modulo division operation that returns the integer remainder of a quotient. Zero indicates that there is no remainder and the two values are evenly divisible.

To find the GCD, start with the smallest operator in the set and divide it into all other operators in the set until a non-integer value is returned (the result $\neq 0$). This is easily accomplished using the modulo division operator. If there are more operators remaining in the set when this occurs, subtract one from the initial divisor and perform the division using the new divisor until a value not equal to 0 is returned or the end of the set is reached. Continue decrementing the value of the divisor by 1 (if not at the end of the set) until all operators are evenly divisible by the divisor or it equals 1. The first value of a divisor that evenly divides all operators in the set is the GCD. The algorithm can be stated as follows:

1. Operators : set. Initially it contains all operators.
2. N := 1.
3. Blocks : set. Initially it is empty.
4. GCD : integer.
5. GCD := smallest period in Operators.
6. Divide GCD into each operator period in Operators until period(operator) Mod GCD ≠ 0 or the end of the set is reached.
7. If the end of the set was not reached, GCD := GCD - 1.
8. If all remainders = 0, GCD is the greatest common divisor.
9. If necessary, repeat Steps 6 - 8 until GCD = 1.
10. Blocks := Blocks ⋃ Operators.
11. Base_period := GCD.

An example of this algorithm will now be presented using the operators introduced in Figure 10 on page 27. For purposes of this example, assume that Operator A has a period of 3, B has a period of 6, and C has a period of 10. These periods may not be consistent with any sort of application but are sufficient to demonstrate the algorithm. The example is presented below:

Step 1. Operators := { A.3, B.6, C.10 } . Recall that the operators were previously sorted by period in ascending order.

Step 2. N := 1.

Step 3. Blocks := { } .

Step 4. GCD := 0.

Step 5. GCD := 3. (smallest period in Operators)

Step 6. 3 Mod 3 = 0, 6 Mod 3 = 0, 10 Mod 3 = 1. Both "quit" conditions are reached since 10 Mod 3 ≠ 0 and C.10 is the last operator in the set.

Step 7. GCD := 3 - 1, (GCD := 2).

Step 8. Not applicable.

Step 9. Repeat Steps 6 - 8 until GCD = 1 or end of set is reached.

Step 6. 3 Mod 2 = 1, since 1 ≠ 0, we skip to Step 7.

Step 7. GCD := 2 - 1 (GCD := 1). Since GCD = 1, skip back to Step 10.

Step 10. Blocks := { } ⋃ { 3, 6, 10 } . (Blocks = { 3, 6, 10 } and has a base period equal to 1)

Step 11. Base_period := 1.

b. Multiple processors

When working in a multiprocessor environment, more than one harmonic block can be constructed. There may be as many harmonic blocks as there are processors. The number of processors available will be known beforehand to the static scheduler so that it does not try to construct more blocks than there are processors. Referring to the definition of a harmonic block given earlier in this chapter, it is easy to identify which operators belong together in a single harmonic block. The base period for any harmonic block is the greatest common divisor of all the operators and in this case, will equal the smallest period of all the operators in the block.

The Build_harmonic_blocks algorithm basically takes the operator with the smallest period from the set of all operators and makes it the base period of the harmonic block. The periods of the remaining operators in the set are then divided by this base period. If the division produces an integer result (no remainder), the operator has a period that is an exact multiple of the base period and is assigned to the harmonic block. As each operator is assigned to the block, it is deleted from the set of all operators. When all operators in the set have been tested, the algorithm repeats and selects the operator with the smallest period to be the base period for the next harmonic block. This new base period is divided into the remaining periods and operators are assigned to the block and deleted from the set as above. This iterative process continues until all operators have been assigned or there have been N - 1 harmonic blocks created where N is the total number of processors available. The last harmonic block will be constructed as in the single processor case described in the previous section with the base period being equal to the GCD of the remaining operators.

Once all the blocks have been constructed, there must be one more pass over them to ensure that an operator that meets the criteria for more than one harmonic block is assigned to the block with the largest base period. It is believed that this will help make the scheduling problem easier. Essentially what can be done is the base period of each harmonic block beginning with the block having the second smallest base period is divided into each operator period in all other harmonic blocks having a smaller base period. If the result of the division is an integer value (remainder of 0), the operator is moved to the harmonic block with the larger base period.

The algorithm can be stated as follows:

1. Operators : set. Initially it contains all operators.
2. Blocks : set. Initially it is empty.

3. N := number of processors.
4. Base_period : integer, initialized to 0.
5. New_block : set. Initially it is empty.
6. While Operators is not empty or the number of elements in blocks is < N - 1, do Steps 7 - 9.
7. Base_period := first operator period in Operators.
8. Each remaining operator in Operators is divided by Base_period using the Mod operation. If period(operator) Mod Base_period = 0, delete operator from Operators and add it to New_block.
9. Blocks := Blocks ⋃ New_block.
10. Repeat Steps 6 - 9 until Operators is empty or the number of elements in Blocks = N - 1.
11. GCD := greatest common divisor of the remaining operators in Operators if Operators is not empty. (use the greatest common divisor algorithm for the single processor case)
12. Base_period := GCD.
13. Divide operators in Operator by Base_period and assign to New_block as in Step 8.
14. Blocks := Blocks ⋃ New_block.
15. Operators should now be empty.
16. Beginning with the element in Blocks having the second smallest base_period, divide this base period into each operator period in all other elements in Blocks having a smaller base period.
17. If period(operator) Mod base_period = 0, subtract that operator from the harmonic block and add it to the harmonic block with the larger base period.
18. Repeat until all base periods have been processed.

Using the same three operators, A, B, and C, the Build_harmonic_blocks algorithm will be demonstrated for the multiprocessor case. Again, this is an oversimplified example but it should illustrate the algorithm adequately. The example is presented below:

Step 1. Operators := { A.3, B.6, C.10 } .

Step 2. Blocks := { } .

Step 3. N := 2. (assumed for this example)

Step 4. Base_period := 0.

Step 5. New_block := { } .

Step 6. Operators is not empty and there are 0 elements in blocks (0 < 1), so we go to Step 7.

Step 7. Base_period := 3.

Step 8. 6 Mod 3 = 0, 10 Mod 3 = 1, New_block := { A.3, B.6 } , Operators := { 10 } .

Step 9. Blocks := { { A.3, B.6 } } .

Step 10. The number of elements in Blocks = 1. 1 = 1 so we go to step 11.

Step 11. Operators is not empty so the GCD is determined using the single processor algorithm. The GCD for { 10 } = 10. (10 Mod 10 = 0).

Step 12. Base_period := 10.

Step 13. 10 Mod 10 = 0, New_block := { C.10 } , and Operators = { } .

Step 14. Blocks := { { A.3, B.6 } } $\cup$ { C.10 } = { { A.3, B.6 } , { C.10 } } .

Step 15. Operators = { } .

Steps 16-18. The element in Blocks having the second largest base period is { 10 } with a base period of 10. Since there is only one other element in Blocks, this step is fairly simple. 3 Mod 10 $\neq$ 0 and 6 Mod 10 $\neq$ 0 so all operators will remain in their respective harmonic blocks.

4. Find_block_length

The final requirement in building the harmonic blocks is to determine their length in units of time. This is important for two reasons. First, once the length is known, another test can be performed to ensure that all operators in the block can be scheduled as dictated by their timing constraints. This is done by multiplying each operator's maximum execution time (MET) by the number of times it is supposed to be scheduled within the block (block length $\div$ operator period). The sum over all the operators must be less than the block length. This can be represented mathematically as

$$\Sigma \text{ MET(i)} * (\text{block length/period(i)}) \leq \text{block length.}$$

This is a necessary though not sufficient condition to ensure that a valid schedule can be constructed.

Second, each harmonic block is itself periodic. The block length determines how often the block will repeat.

The length of a harmonic block is simply the least common multiple (LCM) of all the operators in the block. Z is defined as the LCM of (X,Y) if and only if Z Mod X = 0 and Z Mod Y = 0 and (W Mod X = 0 and W Mod Y = 0) implies W $\geq$ Z. The LCM is computed by taking two periods at a time, multiplying them together, and then dividing this result by the greatest common divisor of the two periods. This result

is then multiplied together with the next period and divided by their GCD until all operators in the set have been processed. The result of this operation on the last pair in the set is the least common multiple of all operators in the set. The algorithm is presented below :

1. GCD, A, B, C, D : integer. All are initialized to zero.
2. A := 1st period.
3. B := 2nd period.
4. C:= A * B.
5. GCD := greatest common divisor of A and B (using the GCD algorithm of section 3).
6. D := C ÷ GCD.
7. A := D.
8. B := next period.
9. Continue until there are no more operators in the set. The LCM equals the result from the final pair of operator periods.

To continue with our example, the block length of the harmonic block containing Operators A, B, and C ({ 3, 6, 10 }) in the single processor case will be computed following the algorithm outlined above. The LCM is shown in the following example:

Step 1. GCD, A, B, C, D := 0.

Step 2. A := 3.

Step 3. B := 6.

Step 4. C := 18.

Step 5. GCD := 3.

Step 6 D := 18 ÷ 3, (D := 6).

Step 7. A := 6.

Step 8. B := 10.

Step 4. C := 60.

Step 5. GCD := 2.

Step 6. D := 30.

Step 7. A := 30.

Step 8. There are no more operators.

Step 9. LCM = 30.

E. SCHEDULE_OPERATORS

The 2nd level data flow diagram for Schedule_operators is shown in Figure 14 on page 43. The operators within each harmonic block are scheduled according to their precedence and period constraints. As each operator is scheduled, a Next_firing_interval is calculated. The lower bound of the Next_firing_interval represents the earliest time at which the operator can be scheduled. The upper bound of the interval is the latest time at which the operator can be scheduled and still meet an end of period deadline.

1. Schedule_next_operator

Schedule_next_operator is where the topologically sorted schedule is combined with the harmonic block schedule. Essentially, the next operator to be scheduled is determined by the value of an actual time, t. As each operator is scheduled, it is given a start time, stop time, and Next_firing_interval. This discussion of scheduling operators will be focused on the single processor case. The primary difference between scheduling operators for single and multiprocessor systems is at what time the first operator within each harmonic block can be scheduled. In the multiprocessor case, instead of automatically being scheduled at time t = 0, the block may have to be shifted in time to allow for precedence relationships between operators in the different blocks.

In the single processor case the first operator is taken from the top of the precedence list and is scheduled to start at time t = 0. Its start time is 0 and the stop time is equal to its MET. The actual time t equals the MET of the first operator. The second operator is then selected from the precedence list. Its start time is equal to the stop time of the previous operator and its stop time is equal to its start time + MET. The actual time is made equal to the stop time. Before scheduling the third and subsequent operators, it is necessary to compare actual time t with the Next_firing_intervals that are calculated when each operator is scheduled. Three cases could exist. First, the actual time may be less than the lower bound of every interval. In this case, the next operator is selected from the precedence list to start at the actual time. A new actual time is computed which is equal to the stop time for this operator (start time + MET). If all of the operators in the precedence list have been scheduled at least once, the next operator to be scheduled is the one with the smallest lower bound. The start time for an operator so scheduled will equal this lower bound creating a gap in the schedule. The actual time t is then computed as before.

The second case is where the actual time t falls within one or more Next_firing_intervals. If it falls within a single interval, that operator is scheduled to

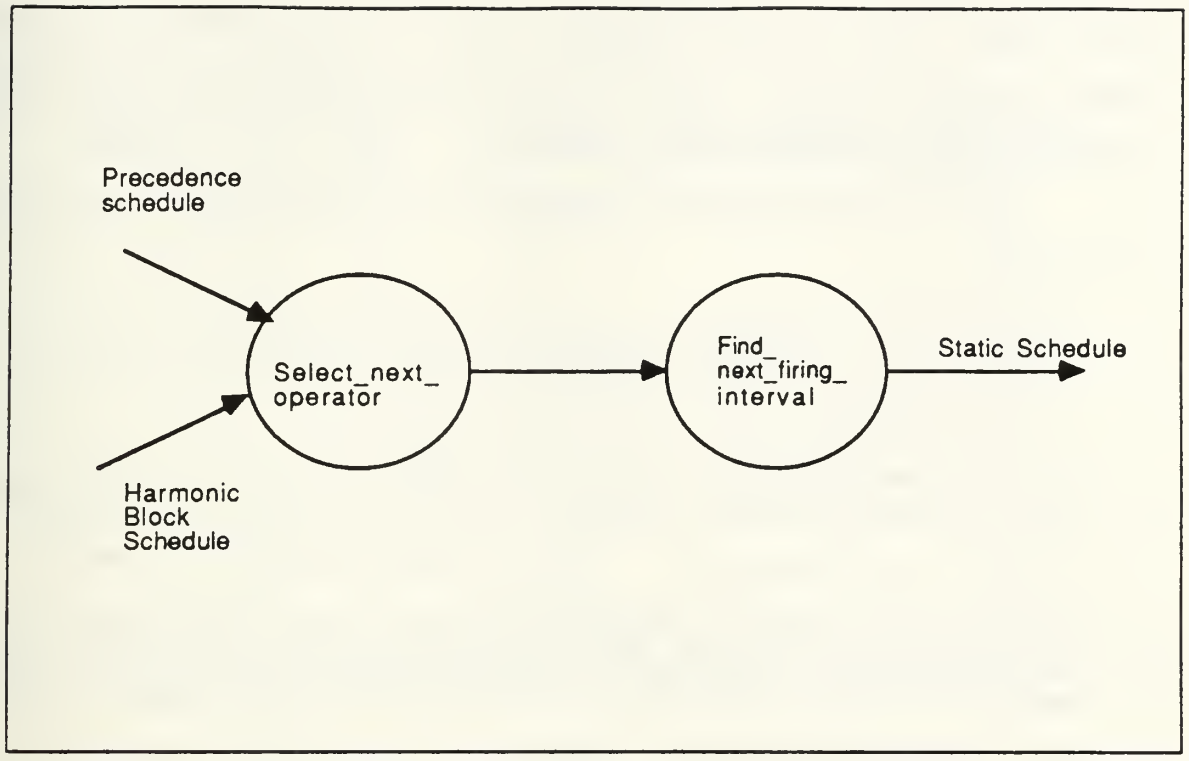


Figure 14. Schedule_operators, 2nd Level Data Flow Diagram

start at actual time t. If the actual time falls within two or more intervals, the operator with the smallest upper bound is chosen to be scheduled next (earliest deadline first).

The third possibility is that the actual time t may be greater than the upper bound of one or more Next_firing_intervals. If this situation occurs, a valid schedule cannot be built since periodic timing constraints within the harmonic block cannot be met.

2. Find_next_firing_interval

As can be gathered from the above discussion, the Next_firing_interval is an important component of Schedule_operators. The formula for calculating the Next_firing_interval can be stated as follows

Next_firing_interval =

[(Start time + period), (Start time + 2period - MET)] .

The lower bound is found by simply adding an operator's period to the time when it is scheduled to start. This ensures that at least the length of one period will pass before the operator is scheduled to fire again. The upper bound on the interval ensures that an operator is scheduled early enough so that it can finish execution prior to the end of

its period. It is calculated by adding twice the period to the start time and subtracting off the MET. If an operator is scheduled to start at a time greater than this upper bound, there is no guarantee that it will finish execution prior to its deadline. This causes the whole static schedule to be invalid.

The entire Schedule_operators algorithm for the single processor case is presented below:

1. Start at time t = 0.
2. Select first operator from the topologically sorted list. Schedule it to start at t = 0 and stop at t = 0 + MET. Actual time t = 0 + MET.
3. Calculate Next_firing_interval based on the formula.
4. Select next operator from the topologically sorted list. Schedule it to start at actual time t and stop at start time + MET.
5. Calculate Next_firing_interval.
6. Compare actual time t with Next_firing_intervals. If lower bound ≤ actual time t ≤ upper bound, schedule that operator. If actual time t falls in two or more Next_firing_intervals, choose the one with the smaller upper bound to meet the earliest deadline first.
7. Calculate Next_firing_interval and replace the old Next_firing_interval with the new one.
8. If actual time t < lower bound of all intervals, select the next unscheduled operator from the topologically sorted list. Schedule it to start and stop as in Step 4 and compute its Next_firing_interval. If all operators have been scheduled at least once, select the next operator to be scheduled as that which has a Next_firing_interval with the lowest bound. (A gap may be created in the schedule)
9. If actual time t > upper bound of any interval, a valid schedule cannot be constructed. Raise an exception and cease execution.
10. Continue scheduling operators until actual time t is greater than the harmonic block length or all operators' Next_firing_intervals have lower bounds greater than the block length.

To continue with our single processor example, Operators A, B, and C have been topologically sorted and must be scheduled in that order. From Figure 10 on page 27, the METs were given as 1, 2, and 2 respectively. Recall that the base period for the harmonic block { 3, 6, 10 } is 1 and the block length (LCM) is 30. Figure 15 on page 45 summarizes the pertinent operator timing constraints and the results of the two additional tests mentioned in Sections D and E. 26 out of 30 time slots will be filled by operators A, B, and C leaving four unused for the dynamic scheduler. The sum of the METs ÷ periods equals .86. This is less than 1 which is the single processor limit. Both of these results indicate the feasibility of a valid static schedule.

Operator	MET	Period	LCM/Period	MET * (LCM/Period)	MET/Period
A	1	3	10	10	.33
B	2	6	5	10	.33
C	2	10	3	6	.2
				26 < 30	.86 < 1

Figure 15. Operator Timing Constraints

Figure 16 on page 46 shows how the steps of the Schedule_operators algorithm were applied to these operators. Operator A was scheduled first since it was at the top of the precedence list. It is scheduled to start at time t = 0 and stop at actual time t = 1. Its Next_firing_ interval was calculated to be [3, 5,] . Operator B is scheduled next based on precedence. It will start at time t = 1 and stop at actual time t = 3. The Next_firing_interval is calculated to be [7, 11] . The actual time t = 3 is compared with the Next_firing_ intervals and matches the lower bound of A's interval. A is therefore scheduled next to start at time t = 3 and stop at time t = 4. The Next_firing_interval is [6, 8] . This actual time t = 4 is compared with the Next_firing_intervals and since it is less than all lower bounds, operator C is selected from the precedence list. C is scheduled to start at time t = 4 and stop at time t = 6. Its Next_firing_interval is [14, 22] . Since there are no additional operators in the precedence list, the remaining schedule is based on the Next_firing_intervals. After each operator is scheduled, actual time t is compared to the Next_firing_intervals only. Following the algorithm, the operator that is selected to be scheduled next is the one that has the earlier deadline. When all operators have Next_firing_intervals with lower bounds that are greater than or equal to the harmonic block length, the scheduling is complete. In the example, this occurs for operator B at stop time t = 25, for operator A at stop time t = 28, and for operator C at stop time t = 30.

Operator	Start	Stop	Next_firing_interval
A	0	1	[3, 5]
B	1	3	[7, 11]
A	3	4	[6, 8]
C	4	6	[14, 22]
A	6	7	[9, 11]
B	7	9	[13, 17]
A	9	10	[12, 14]
A	12	13	[15, 17]
B	13	15	[19, 23]
A	15	16	[18, 20]
C	16	18	[26, 34]
A	18	19	[21, 23]
B	19	21	[25, 29]
A	21	22	[24, 26]
A	24	25	[27, 29]
B	25	27	[31, 35]
A	27	28	[30, 28]
C	28	30	[38, 46]

Figure 16. Example of Scheduling a Harmonic Block

The final static schedule is depicted graphically in Figure 17 on page 47. Note that there are in fact four unused time slots and these occur at times 10 - 12 and 22 - 24.

This concludes the initial design for the static scheduler. To summarize briefly, this static scheduler extracts the timing information necessary to construct a schedule directly from the PSDL prototype source file. Non-time critical operators are separated and sent to the dynamic scheduler for run-time scheduling. The remaining time critical operators are sorted topologically to build a schedule based on operator precedence. In

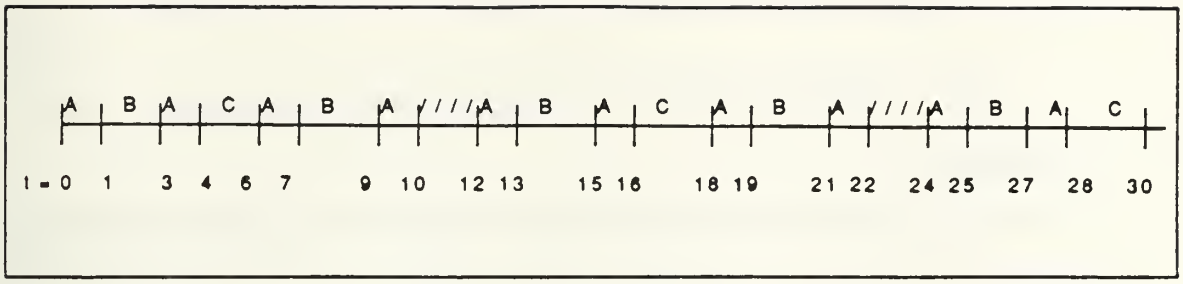


Figure 17. Example Static Schedule

addition, a separate schedule based on timing constraints is constructed. Finally, the two schedules are merged together to form one static schedule that will meet both precedence and timing constraints.

The PSDL static scheduler is designed to be "generic" in that it should be able to schedule operators in a PSDL prototype for any type of hard real-time system. Recommendations for further research on the static scheduler are contained in Chapter IV.

IV. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY

This thesis has provided an introduction to two software engineering methodologies, the traditional life cycle and rapid prototyping. In particular, the rapid prototyping methodology was discussed as a promising approach to the development of software more efficiently and at less cost. The Computer Aided Prototyping System (CAPS) was introduced as a software engineering tool that is currently being designed. This tool will enable software designers to exploit rapid prototyping to its fullest by automating the construction of executable prototypes. The execution support system is the component within the CAPS that makes the prototype, written in the Prototype System Description Language (PSDL), executable. The major contribution of CAPS to the advancement of software engineering technology lies in the fact that the executable prototypes can be automatically generated by the use of specifications and reusable software components.

A review of the current literature has underscored not only the need for this type of software engineering tool but also that efforts in this area are not being duplicated. Most of the research in the design of hard real-time systems focuses on the hardware aspects of timing constraints or the simulation of the timing specifications by logic programming. CAPS, via PSDL, simulates the hardware aspects of hard real-time systems (such as sensors and probes) thereby modeling a system architecture as well as executing the prototype with practical computation times.

The contribution of this thesis to CAPS research was the development of a conceptual design for the static scheduler, one of the components in the CAPS Execution Support System. The design is the pioneer prototype design for the static scheduler. This design should allow operators from any type of software system, even those with control based on data flow, to be scheduled in a way that meets all critical timing constraints.

B. FURTHER RESEARCH

Because this is a pioneer design, further research is required for implementation and identification of possible weaknesses. Without an executable version of the static scheduler, it is difficult to identify all possible software design contingencies. In addition

to these unknown problems, this author recommends continued work in the following areas:

Implementation of the Static Scheduler,

Handling Simple Validity Checks,

Implementation of the Execution Support System Interfaces,

Handling Feasibility Tests, and

Scheduling Operators in a Multiprocessor Environment.

1. Implementation of the Static Scheduler

As conceptualized, the implementation will be in Ada. A guide for accomplishing the implementation is contained in [Ref. 9].

2. Handling Simple Validity Checks

As it is currently designed, the static scheduler performs some validity checks on the timing information that is provided by the system designer and notifies the designer if any information is invalid. Execution of the prototype cannot continue without the designer altering the timing information as necessary and running the program again. It may be possible for the static scheduler not only to identify the problem, but also to correct it. The scheduler would have to pick a feasible value for whatever attribute is in question based on worst case criteria. The designer would still have to be notified of the situation; the difference is that execution would not be suspended.

The prototype design presented in this thesis has assumed that all timing constraints for an operator have been supplied by the designer. A more sophisticated design could handle those instances where some required information is missing. Again, the static scheduler could assign a value based on some worst case criterion.

3. Implementation of the Execution Support System Interfaces

Chapter I briefly touched on the relationship between the three components of the Execution Support System. As was shown in Figure 5 on page 11, the static scheduler interfaces with the dynamic scheduler. Since the algorithms for both schedulers were designed independently, there may need to be some modifications made to ensure proper execution. For instance, when the static scheduler has finished extracting operator information from the PSDL source file, it passes a separate text file to the dynamic scheduler containing information about the non-time critical operators in a prototype. Both schedulers will have to agree on a format for the file as well as what information the file will specifically contain. The same formatting issue could apply to the static schedule itself which is also passed to the dynamic scheduler. The dynamic

scheduler is also responsible for instantiating the static scheduler at run time. This is an implementation problem rather than a design problem but it still must be addressed to ensure proper interfacing.

4. Handling Feasibility Tests

Two tests were described in Chapter III which could be done to indicate the feasibility of constructing a valid schedule once all operators had periods and were assigned to harmonic blocks. As with the simple validity checks, in the event it is determined that a valid static schedule in not feasible, program execution is discontinued. It is also possible in this situation to modify some timing constraints for the purpose of constructing the schedule rather than requiring the system designer to input all corrections. An exception would still be raised to notify the designer of the problem and what actions were taken to correct it. Only if attempts to modify timing information prove too difficult should the program be allowed to cease execution prior to completion.

5. Scheduling Operators in a Multiprocessor Environment

The algorithm for scheduling operators within harmonic blocks that was presented in Chapter III is primarily for use in a single processor environment. It should only require slight adjustments to this algorithm to make it suitable for use with multiprocessor systems. One of the adjustments that is necessary is in the algorithm for scheduling the first operator in each harmonic block. Even though each harmonic block is a separate scheduling problem, there will be precedence relationships between some of the operators in separate blocks. For this reason, the first operator in every harmonic block will not necessarily be able to be scheduled to start at time $t = 0$. The algorithm needs to incorporate this possible situation.

C. APPLICATIONS TO DOD TELECOMMUNICATIONS SOFTWARE DESIGN

It is virtually impossible with today's technology to separate telecommunications from computers. Command and Control requirements in the Navy and the DoD often stipulate a need for real-time or very near real-time communications capabilities. Today's communications networks are very complex and cover extensive geographical areas. The software necessary for the reliable operation of these networks is also very complex, often requiring several thousands of lines of code and many years to implement.

In addition to size and development time, there are some additional challenges to software design presented by telecommunications systems. First, many

telecommunications implementations must meet stringent real-time requirements. A digitized or packetized voice system is an excellent example. Routing of the voice packets must be done so that the receiver is provided with an intelligent signal. Packet processing, therefore, is subject to hard real-time constraints. In a switched telephone network, the switch is a real-time system in which the timing of events plays an important role. For instance, a caller should receive a dial tone within a specified period of time after picking up the receiver. A caller is often required to dial the first number within a certain period of time after receiving the dial tone before a warning tone is heard. A caller should receive a ringback tone no more than a specified time after the receiving telephone rings. All of these are real-time constraints which must be adhered to if the system is to function consistently and properly.

Second, communications protocol standards tend to be incomplete and sometimes inconsistent. The Computer Aided Prototyping System can be used to prototype communications protocols and validate them. Potential problems such as deadlock can be determined early on before a great deal of time and money has been spent.

Third, communications software must be interoperable across a variety of incompatible hardware and software environments. This is primarily true for wide area networks such as the Defense Data Network (DDN) but it can also hold for local area networks (LANs). Both types of networks can comprise equipment from multiple vendors. Protocol compatibility is a very important problem. The software interfaces or emulators can be designed using the CAPS methodology to identify incompatibility problems early on by simulating the properties of the hardware components of a network. This is an example of where the CAPS can be used in the design of any large software system, not just those with hard real-time constraints. Interoperability problems may also result from equipment that is obsolete or that is poorly documented. These can also be easily overcome by using a software design tool such as the CAPS.

Fourth, communications software must be updated as protocol ambiguities are recognized and changes are made. For software designed using the CAPS tool this is very easy to do. One of the advantages of using an automated prototyping tool is that the modules of code used in the prototype can often be used "as is" in the final software product. Another advantage is that CAPS and PSDL stress good modularity of software components. Taken together, these result in changes or corrections to existing systems to be very easy to implement. In addition, since the prototype design is maintained in

the prototype database, the effect of the updates on system performance can be prototyped quickly and easily.

D. CONCLUSIONS

The automatic generation of hard real-time system prototypes is feasible. The work done to date on the Computer Aided Prototyping System is beginning to gain attention as a possible solution to today's software design crisis. CAPS will be written in Ada and its focus is on the design of large software systems (with or without real-time constraints) written in Ada. Since the Department of Defense has indicated that all newly developed systems should be implemented in Ada, this tool will be indispensible. The specification language (PSDL) is much easier to learn and work with than Ada. Once the reusable software base containing software components in Ada has matured, designers will be required to hand code only a small fraction of their systems in Ada which will save a great deal of time.

A recent Secretary of the Navy instruction requires software rapid prototyping to be used in the acquisition of software-intensive Command and Control information systems [Ref. 23]. The Computer Aided Prototyping System with its emphasis on the rapid prototyping methodology will make it substantially easier for the Navy to conform to this new software acquisition policy.

Finally, with today's emphasis in the DoD and DoN on cost control and budgetary awareness, CAPS can contribute significantly towards lowering communications software development and maintenance costs.

APPENDIX A. PSDL GRAMMAR

This appendix contains the entire PSDL grammar. Optional items are enclosed in [square brackets] and items that may appear zero or more times appear in { braces } . Terminal symbols appear in "Double quotes".

```
psdl = { component }

component = data_type | operator

data_type = "type" id type_spec type_impl

operator = "operator" id operator_spec operator_impl

type_spec = "specification" [ type_decl ]
            { "operator id operator_spec }
            [ functionality ] "end"

operator_spec = "specification" interface
                [ functionality ] "end"

interface = { attribute [ reqmts_trace ] }

attribute = generic_param
          | input
          | output
          | states
          | exceptions
          | timing_info

generic_param = "generic" type_decl

input = "input" type_decl

output = "output" type_decl

states = "states" type_decl "initially" expression_list

exceptions = "exceptions" id_list

id_list = id { "," id }

timing_info = [ "maximum execution time" time ]
              [ "minimum calling period" time ]
              [ "maximum response time" time ]

time = integer [ unit ]
```

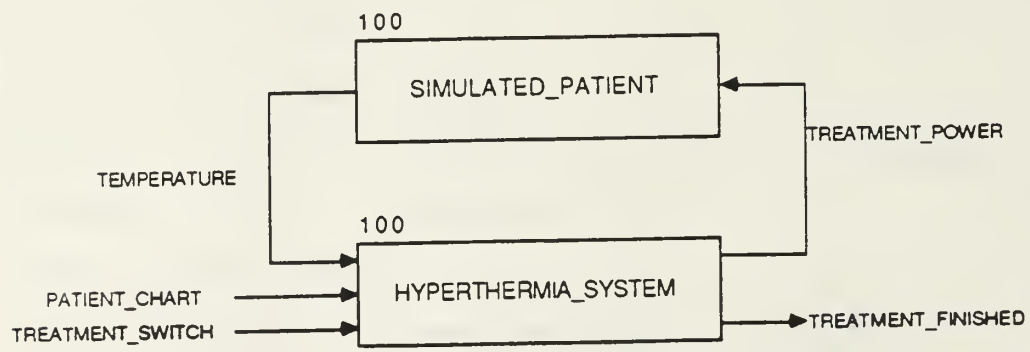
```
unit = "microsec" | "ms" | "sec" | "min" | "hours"

reqmts_trace = "by requirements" id_list


functionality = [ keywords ]
                [ informal_desc ]
                [ formal_desc ]

keywords = "keywords" id_list

informal_desc = "description" " { " text " } "

formal_desc = "axioms" " { " text " } "

type_impl = "implementation" "Ada" id
          | "implementation" type_name
            { "operator id operator_impl } "end"

operator_impl = "implementation" "Ada" id
              | "implementation" psdl_impl

psdl_impl = data_flow_diagram
            [ streams ]
            [ timers ]
            [ control_constraints ]
            [ informal_desc ]
            "end"

data_flow_diagram = "graph" { link }

link = id "." op_id "->" id

op_id = id [ ":" time ]

streams = "data stream" type_decl

type_decl = id_list ":" type_name { "," id_list ":" type_name }

type_name = id | id " [ " type_decl " ] "

timers = "timer" id_list

control_constraints = "control constraints" { constraint }

constraint = "operator" id
             [ "triggered" [ trigger ] [ "if" predicate ]
                           [ reqmts_trace ] ]
             [ "period" time [ reqmts_trace ] ]
             [ "finish within" time [ reqmts_trace ] ]
             { "output" id_list "if" predicate
               [ reqmts_trace ] }
             { "exception" id [ "if" predicate ]
               [ reqmts_trace ] }
```

```
            { timer_op id [ "if" predicate ]
              [ reqmts_trace ] }

timer_op = "reset timer" | "start timer" | "stop timer"

trigger = "by all" id_list
        | "by some" id_list

predicate = "not" predicate
          | predicate "and" predicate
          | predicate "or" predicate
          | expression
          | id ":" id_list

expression = constant
           | id
           | type_name "." id "(" expression_list ")"

expression_list = [ expression { "," expression } ]
```

APPENDIX B. PSDL HYPERTHERMIA EXAMPLE

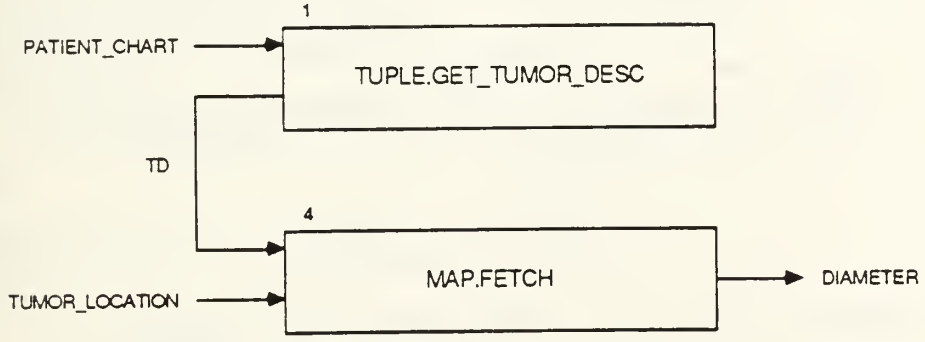
The following is an example of a PSDL prototype for the computer software subsystem of a real-time system for the treatment of brain tumors. The prototype contains six operators, the specifications and implementations of which are presented in hierarchical order, with the highest level composite operators appearing before their respective decompositions.

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient _chart:  medical_history,
    treatment_switch:  boolean
 OUTPUT treatment_finished:  boolean
 STATES temperature:  real
  INITIALLY 37.0
 DESCRIPTION
 { The brain tumor treatment system kills tumor cells
   by means of hyperthermia induced by microwaves.
 }
END

IMPLEMENTATION
 GRAPH
```



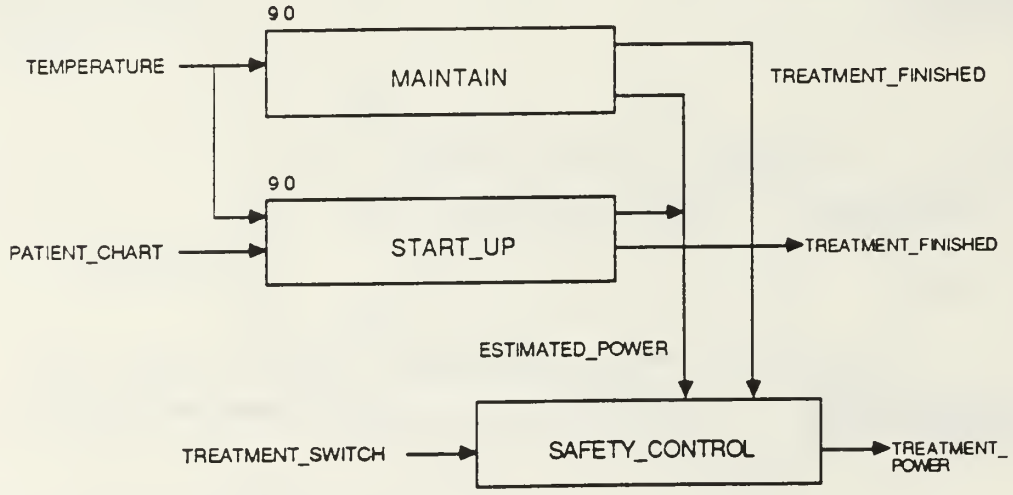

```
 DATA STREAM treatment_power:  real
 CONTROL CONSTRAINTS
  OPERATOR hyperthermia-system
   PERIOD 200 BY REQUIREMENTS shutdown
  OPERATOR simulated_patient
   PERIOD 200
 DESCRIPTION { paraphrased output } END
```

```
TYPE medical_history
SPECIFICATION
 OPERATOR get_tumor_diameter
  SPECIFICATION
   INPUTS patient_chart:  medical_history,
      tumor_location:  string
 OUTPUTS diameter:  real
 EXCEPTIONS no_tumor
 MAXIMUM EXECUTION TIME 5 ms
 DESCRIPTION
 { Returns the diameter of the tumor at a given location,
   produces an exception if no tumor at that location.
 }
 END


 KEYWORDS patient_charts, medical_records, treatment records,
    lab records
 DESCRIPTION
 { The medical history contains all of the disease and
   treatment information for one patient.  The operations
   for adding and retrieving information not needed by
   the hyperthermia system are not shown here.
 }
END

IMPLEMENTATION
 tuple [tumor_desc:  map[from:  string, to:  real], ... ]

 OPERATOR get_tumor_diameter
  IMPLEMENTATION
   GRAPH
```




```
    DATA STREAM td:  tumor_description
    CONTROL CONSTRAINTS
     OPERATOR map.fetch
      EXCEPTION no_tumor IF not(map.has(tumor_location, td))
   END
```

```
END


OPERATOR hyperthermia_system
SPECIFICATION
 INPUT temperature:  real, patient_chart:  medical_history,
    treatment_switch:  boolean
 OUTPUT treatment_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 100 ms
  BY REQUIREMENTS temperature_tolerance
 MAXIMUM RESPONSE TIME 300 ms
  BY REQUIREMENTS shutdown
 KEYWORDS medical_equipment, temperature_control,
    hyperthermia, brain_tumors
 DESCRIPTION
  { After the doctor turns on the treatment switch, the
    hyperthermia system reads the patient's medical record
    and turns on the microwave generator to heat the tumor
    in the patient's brain.  The system controls the power
    level to maintain the hyperthermia temperature of
    42.5 degrees C. for 45 minutes to kill the tumor cells.
    When the treatment is over, the system turns off the
    power and notifies the doctor.
  }
 END

IMPLEMENTATION
 GRAPH
```




```
 DATA STREAM estimated_power:  real
 TIMER treatment_time
  CONTROL CONSTRAINTS
   OPERATOR start_up
    TRIGGERED IF temperature < 42.4
     BY REQUIREMENTS maximum_temperature
    STOP TIMER treatment_time
```

```
    RESET TIMER treatment_time IF temperature <=37.0

   OPERATOR maintain
    TRIGGERED IF temperature >=42.4
     BY REQUIREMENTS maximum_temperature
    START TIMER treatment_time
     BY REQUIREMENTS treatment_time, temperature_tolerance
    OUTPUT treatment_finished IF treatment_time >= 45 min
     BY REQUIREMENTS treatment_time

END


OPERATOR start_up
SPECIFICATION
 INPUT patient_chart:  medical_history, temperature:  real
 OUTPUT estimated_power:  real, treatment_finished:  boolean
  BY REQUIREMENTS startup_time
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { Extracts the tumor diameter from the medical history and
   uses it to calculate the maximum safe treatment power.
   Estimated power is zero if no tumor is present.  The
   treatment finished is true only if no tumor is present.
 }
END

IMPLEMENTATION Ada start_up
END


OPERATOR maintain
SPECIFICATION
 INPUT temperature:  real
 OUPUT estimated_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { The power is controlled to keep the power between 42.4
   and 42.6 degrees C.
 }
END

IMPLEMENTATION Ada maintain
END


OPERATOR safety_control
SPECIFICATION
 INPUT treatment_switch, treatment_finished:  boolean
    estimated_power:  real
 OUTPUT treatment_power:  real
  BY REQUIREMENTS shutdown
 MAXIMUM EXECUTION TIME 10 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
```

```
 { The treatment power is equal to the estimated power
   if the treatment switch is true and treatment finished
   is false.  Otherwise the treatment power is zero.
 }
END

IMPLEMENTATION Ada start_up
END



WITH medical_history_package; USE medical_history_package;
PROCEDURE start_up(patient_chart:  IN medical_history;
        temperature:  IN real;
        estimated_power:  OUT real;
        treatment_finished:  OUT boolean ) IS


  diameter:  real;
  k:  constant real  :=0.5;
BEGIN
  diameter  :=get_tumor_diameter(patient_chart, "brain_tumor");
  estimated_power  :=k * diameter**2
  treatment_finished  :=false;
EXCEPTION
 WHEN no_tumor=>
  estimated_power  :=0.0
  treatment_finished  :=true;
```

LIST OF REFERENCES

1. Booch, G., *Software Engineering with Ada®*, 2nd ed., Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA 1987.

2. Berzins, V., and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada®*, Addison-Wesley Publishing Co., Inc., 1988.

3. Luqi, *Research Aspects of Rapid Prototyping*, Tech. Rep. NPS52-87-006, Naval Postgraduate School, Monterey, CA, 1987.

4. Luqi and Ketabchi, M., *A Computer Aided Prototyping System*, Tech. Rep. NPS52-87-011, Naval Postgraduate School, Monterey, CA, 1987 and in *IEEE Software*, pp. 66-72, March 1988.

5. Luqi and Berzins, V., *Rapid Prototyping of Real-Time Systems*, Tech. Rep. NPS52-87-005, Naval Postgraduate School, Monterey, CA, 1987.

6. Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real Time Software", to appear in *IEEE Trans on Software Engineering*, 1988.

7. Luqi, *Execution of Real-Time Prototypes*, Tech. Rep. NPS52-87-012, Naval Postgraduate School, Monterey, CA, 1987 and in *ACM First International Workshop on Computer-Aided Software Engineering*, Cambridge, MA, vol. 2, pp. 870-884, May 1987.

8. Moffitt, Charlie R., *A Language Translator for a Computer Aided Prototyping System*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1988.

9. Janson, Dorothy M., *A Static Scheduler for the Computer Aided Prototyping System: An Implementation Guide*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1988.

10. Eaton, Susan L., *A Dynamic Scheduler for the Computer Aided Prototyping System (CAPS)*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1988.

11. Dasarathy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them", *IEEE Proc of the Real-Time Systems Symposium*, Los Angeles, CA, pp. 197-204, December 7-9, 1982.

12. Mok, Aloysius K. and Sutanthavibul, S., "Modeling and Scheduling of Dataflow Real-Time Systems", *IEEE Proc of the Real-Time Systems Symposium*, San Diego, CA, pp. 178-187, December 3-6, 1985.

13. Bic, L., *A Process-Oriented Model for Efficient Execution of Dataflow Programs*, Tech. Rep. 86-23, Univ. of California, Irvine, CA, 1986.

14. Mok, Aloysius K., "A Graph-Based Computation Model for Real-Time Systems", *Proc of the IEEE International Conference on Parallel Processing*, Pennsylvania State Univ., PA, pp. 619-623, August 20-23, 1985.

15. Mok, Aloysius K., "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Proc of the Real-Time Systems Symposium*, Austin, TX, pp. 125-134, December 4-6, 1984.

16. Mok, Aloysius K., "The Design of Real-Time Programming Systems Based on Process Models", *IEEE Proc of the Real-Time Systems Symposium*, Austin, TX, pp. 5-17, December 4-6, 1984.

17. Leinbaugh, Dennis W., "Guaranteed Response Times in a Hard-Real-Time Environment", *IEEE Trans on Software Engineering*, vol. SE-6, no. 1, pp. 85-91, January 1980.

18. Abbot, C., "Intervention Schedules for Real-Time Programming", *IEEE Trans on Software Engineering*, vol. SE-10, no. 3, pp. 268-274, May 1984.

19. Ladkin, P., "Specification of Time Dependencies and Synthesis of Concurrent Processes", *Proc of the 9th International Conference on Software Engineering*, Monterey, CA, pp. 106-115, March 30-April 2, 1987.

20. Jahanian, F., and Mok, Aloysius K., "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Trans on Software Engineering*, vol. SE-12, no. 9, pp. 890-904, September 1986.

21. Jahanian, F., and Mok, Aloysius K., "A Graph-Theoretic Approach for Timing Analysis in Real Time Logic", *IEEE Proc of the Real-Time Systems Symposium*, New Orleans, LA, pp. 98-108, December 2-4, 1986.

22. Plattner, B., "Real-Time Execution Monitoring", *IEEE Trans on Software Engineering*, Vol. SE-10, No. 6, pp. 756-764, November 1984.

23. Department of the Navy, SECNAV INSTRUCTION 5200.37, *Acquisition of Software-Intensive C² Information Systems*, January 5, 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Office of the Chief of Naval Operations Code OP-941 Washington, DC 20350-2000	1
4.	Office of the Chief of Naval Operations Code OP-945 Washington, DC 20350-2000	1
5.	Commander, Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue N. W. Washington, DC 20394-5290	1
6.	Naval Telecommunications System Integration Center Naval Communications Unit, Washington Washington, DC 20397-5340	1
7.	Ada® Joint Program Office OUSDRE(R&AT) The Pentagon Washington, DC 20301	1
8.	Space and Naval Warfare Systems Command Attn: Dr. Knudsen, Code PD 50 TD Washington, DC 20363-5100	1
9.	Chief of Naval Research Office of the Chief of Naval Research Attn: CDR Michael Gehl, Code 1224 Arlington, VA 22217-5000	1
10.	Professor Luqi, Code 52Lq Naval Postgraduate School Monterey, CA 93943	1
11.	MAJ John B. Isett, USAF, Code 54Is Naval Postgraduate School Monterey, CA 93943	1

12. Professor D. C. Boger, Code 54Bo — 1
Naval Postgraduate School
Monterey, CA 93943

13. Commander, Naval Security Group Command — 3
Attn: LT Joanne T. O'Hern, Code G30
3801 Nebraska Avenue, N.W.
Washington, DC 20390

14. Defense Communications Agency — 1
Attn: LT Susan L. Eaton, Code B531
Washington, DC 20305

15. LT Charlie R. Moffitt — 1
Department Head Class #104
SWOSCOLOM, Bldg. 446
Newport, RI 02841-5012

16. LT Dorothy M. Janson — 1
USCINCEUR Headquarters
General Delivery
APO New York, NY 09128-4209

17. Naval Sea Systems Command — 1
Attn: CAPT Joel Crandall
National Center #2, Suite 7N06
Washington, DC 22202

18. Office of the Secretary of Defense — 1
Attn: CDR Barber
The Star Program
Washington, DC 20301

19. Navy Ocean System Center — 1
Attn: Lindwood Sutton, Code 423
San Diego, CA 92152-5000

20. RADC COES — 1
Attn: LT Kevin Benner
Griffiss Air Force Base
New York, NY 13441-5700

21. MAJ Mike Dolezal — 1
Director, Development Center
MCDEC
Quantico, VA 22134-5080